

# **Optimally Solving a Rubik's Cube Using Vision and Robotics**

Le Thanh Hoang

Imperial College London  
Department of Computing

Supervised by Andrew Davison

June 15, 2015

## Abstract

The Rubik's Cube, often referred to as 'The Cube', is a puzzle that has troubled many for over 40 years. A puzzle that most people keep in their drawer gathering dust as their previous attempts at solving it have only ended in mindnumbing frustration. With 43,252,003,274,489,856,000 possible combinations in its state space, the cube has a rich and deep mathematical theory attached to it. As daunting as this number may seem, we know that the maximum number of turns required to solve any scrambled state is just 20. We call this: God's Number.[18]

Whilst it is well known that the fastest human solvers need just a few seconds to solve this puzzle, their solutions are far from optimal. In fact, the best human speedsolvers use on average 50-60 moves per solve, simply because a human does not know the entire solution to the cube by just looking at it. In essence, they must solve the cube section by section by putting each colour where it belongs. What humans lack in insight, they make up for in dexterity. The best human speedsolvers can turn up to 10 faces per second.

We take a different approach to solving the cube by using three major components: A vision system that is able to accurately track the cube and its colours using a Smartphone camera so that we can read its state, an algorithm that can find a solution to most cube states in just 22 turns or under, and a robot that is able to reliably turn and solve the cube.

We are able to point our smartphone's camera at each face regardless of background, lighting colour or cube position within the camera frame in order to read any cube state. We are also able to intelligently search through the 43 quintillion combinations to find a close to optimal (sometimes even optimal) solution. Thanks to this, our robot is able to, on average, solve the cube in just 74 seconds.

### **Acknowledgements**

I would like to thank my supervisor, Professor Andrew Davison, for his support throughout the project, his guidance and ideas in the vision and robotics components, funding the Lego and providing MindStorm kits as well as Raspberry Pis and BrickPis. I'd also like to thank Julia Wei for her support, being an excellent proofreader and for providing some suggestions for the robot design. Finally, I would like to thank my parents for supporting me throughout my life.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Motivation and Aims . . . . .	5
1.2	Contributions . . . . .	5
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Fundamental Structure . . . . .	7
2.1.1	Rubik's Cube Jargon . . . . .	7
2.1.2	Face Notation . . . . .	8
2.1.3	Move Notation . . . . .	8
2.1.4	Rotation Notation . . . . .	10
2.1.5	Cube Notation . . . . .	10
2.1.6	Cubie Notation . . . . .	11
2.1.7	Singmaster Notation . . . . .	12
2.2	The Mathematics . . . . .	13
2.2.1	Laws & Lemmas . . . . .	13
2.2.2	Problem space . . . . .	15
2.2.3	Group Theory . . . . .	16
2.2.4	Numbering Schemes . . . . .	17
2.3	Existing Optimal Algorithms . . . . .	19
2.3.1	The Obvious Algorithm: Brute Force . . . . .	19
2.3.2	The First Real Attempt: Thistlethwaite's Algorithm . . . . .	20
2.3.3	A Different Approach: Korf's Algorithm . . . . .	22
2.3.4	Improving Thistlethwaite's Algorithm: Kociemba's Algorithm . . . . .	24
2.3.5	Why Not Human Algorithms? . . . . .	25
2.4	Existing Visioning Systems . . . . .	26
2.4.1	Colour Schemes . . . . .	26
2.4.2	Hardware . . . . .	27
2.4.3	Object tracking . . . . .	28
2.4.4	Colour balancing . . . . .	29
2.5	Existing Robots . . . . .	31
2.5.1	MindCuber . . . . .	31
2.5.2	JPBrown's CubeSolver . . . . .	31
2.5.3	Cubestormer . . . . .	32
2.6	PID Controller . . . . .	33
2.6.1	Open loop vs closed loop . . . . .	33
2.6.2	What is PID specifically? . . . . .	33
2.6.3	What's so great about PID? . . . . .	34
<b>3</b>	<b>Design</b>	<b>35</b>
3.1	Overall Design . . . . .	35
3.1.1	Algorithm Design . . . . .	35
3.1.2	Vision Design . . . . .	36

3.1.3	The Robot Design . . . . .	36
3.1.4	Summary of Design . . . . .	37
<b>4</b>	<b>Implementation</b>	<b>39</b>
4.1	Korf's Algorithm . . . . .	39
4.1.1	Cube representation . . . . .	39
4.1.2	Heuristic generation . . . . .	41
4.1.3	Improvements . . . . .	45
4.1.4	HPPC Java Library . . . . .	48
4.2	Kociemba's algorithm . . . . .	48
4.2.1	Coordinate Labelling . . . . .	48
4.3	Combining Kociemba's and Korf's . . . . .	49
4.3.1	Time to Solve Estimation . . . . .	50
4.4	Searching for shortest number of robot moves . . . . .	50
4.4.1	Dynamic Costing . . . . .	50
4.4.2	Reducing the branching factor . . . . .	50
4.4.3	Search speed . . . . .	51
4.5	Vision System . . . . .	52
4.5.1	Cube Recognition . . . . .	52
4.5.2	Recognising Colour . . . . .	56
4.6	Robot . . . . .	59
4.6.1	Hardware Design . . . . .	59
4.6.2	Movements . . . . .	62
4.6.3	Software . . . . .	63
<b>5</b>	<b>Evaluation</b>	<b>66</b>
5.1	Vision . . . . .	66
5.1.1	Vision accuracy . . . . .	66
5.1.2	Vision limitations . . . . .	68
5.2	Algorithm . . . . .	70
5.2.1	Algorithm speed . . . . .	70
5.2.2	Algorithm solution length . . . . .	71
5.2.3	Algorithm Summary . . . . .	72
5.3	Robot . . . . .	72
5.3.1	Robot Accuracy . . . . .	72
5.3.2	Robot Speed and TPS . . . . .	73
5.3.3	Robot Limitations . . . . .	74
5.3.4	Robot Summary . . . . .	74
5.3.5	System . . . . .	74
<b>6</b>	<b>Conclusions</b>	<b>75</b>
6.1	Future work . . . . .	75
6.1.1	Improving the Algorithm . . . . .	75
6.1.2	Improving vision . . . . .	76
6.1.3	Improving the Robot . . . . .	76
<b>A</b>	<b>System User Guide</b>	<b>80</b>
A.1	Prerequisites . . . . .	80
A.1.1	Hardware requirements . . . . .	80
A.1.2	Setup . . . . .	80
A.2	Using the system . . . . .	80
A.2.1	Reading the state . . . . .	80
A.2.2	Find a solution . . . . .	82
A.2.3	Solving the cube . . . . .	83

# List of Figures

2.1	Labelled Cube	8
2.2	Face Notations	8
2.3	Move Notation Table	9
2.4	Cube Rotations	10
2.5	Cube Net	11
2.6	Cubie Notation	11
2.7	Illegal States	13
2.8	Corner states	14
2.9	Illegal state corner flip	14
2.10	Search Tree	19
2.11	Nodes generated at each depth	19
2.12	A visual representation of each group	21
2.13	Thistlethwaite's group transition size	22
2.14	Thistlethwaite's group transition worst case	22
2.15	Iterative Deepening vs IDA*	23
2.16	HSV Colour Wheel	26
2.17	HSV Saturation Demo	27
2.18	An example of an RGB sensor implementation	27
2.19	We wish to determine edges of this	28
2.20	Getting the first derivative	29
2.21	Second Derivative	29
2.22	MindCuber	31
2.23	CubeSolver	32
2.24	Open vs Closed Feedback loop	33
3.1	The insides of a servo motor	36
3.2	Overview of system through each stage	38
4.1	Corner labelling	40
4.2	Edge labelling	40
4.3	Move method	41
4.4	Main Corner generation loop	43
4.5	Main IDA* loop	44
4.6	Search function	45
4.7	How we split our search space	47
4.8	Contrast and Brightness Adjustment	52
4.13	Finding Stickers	55
4.14	Finding Stickers	55
4.15	HSV colour thresholds	56
4.16	Yellow vs white light	57
4.18	Two different views of the claw	59
4.19	Turn circle of the cube	60
4.20	Two different views of the claw	60

4.21 Overview: Birdseye view . . . . .	61
4.22 Configuration by Arm vs Job . . . . .	62
4.23 move method . . . . .	63
4.24 Moves . . . . .	64
4.25 An example of the whole protocol . . . . .	64
4.26 The rotational gear ratio . . . . .	65
4.27 Degree rotation table . . . . .	65
5.1 Test Results For Vision . . . . .	67
5.2 Sample images of each light scenario . . . . .	68
5.3 A rare case . . . . .	69
5.4 Speed of algorithms in Milliseconds . . . . .	70
5.5 Solution Lengths . . . . .	71
5.6 A perfect alignment . . . . .	72
5.7 Number of face turns and cube rotations until failure . . . . .	73
5.8 Solve times table . . . . .	73
A.1 U Face . . . . .	81
A.2 F Face . . . . .	81
A.3 D Face . . . . .	81
A.4 R Face . . . . .	82
A.5 B Face . . . . .	82
A.6 L Face . . . . .	82
A.7 First Connection Failure . . . . .	83

# Chapter 1

## Introduction

### 1.1 Motivation and Aims

Rubik's Cubes are dead. A mechanical puzzle from the 20th Century in the 21st Century world of computers. However, there is more to the cube than meets the eye. Although to some, it may just be a mindless pastime, to us it presents many interesting challenges in both the underlying Mathematical theory and as a Computer Science search problem. In this project we explore the deep and rich mathematics behind the Rubik's Cube and show how we can exploit this to search for close to optimal solutions using Kociemba's algorithm[14]. We also show how we can intelligently search for optimal solutions using Korf's algorithm[15].

Our end goal is to build a system that is able to read the state of the Rubik's Cube reliably, find a solution and then solve the cube using a robot. We want to demonstrate the challenges associated with building puzzle solving robots. From the real world challenges in vision and robotics down to the theoretical challenges that lie in the search space, this is not a trivial task.

The accuracy needed by the robot should not be overlooked. An error of millimeters from a perfectly aligned 90 degree turn on any face can lead to a disaster for turning adjacent faces. Likewise, the complexity of the vision system should not be underestimated - a single wrongly recognised colour leads to a completely different cube state. As mentioned earlier, finding an optimal solution is difficult in a state space of size  $4.3 * 10^{19}$ . In comparison, a fifteen sliding tile puzzle has only  $10^{13}$  possible states. Unlike many other projects, we are not trying to solve a problem. Instead, we are trying to detail and demonstrate techniques and challenges that are often initially overlooked by those who try to build similar systems.

### 1.2 Contributions

Although Rubik's cube solving robots already exist, they are very rarely documented in any detail at all. In fact, the world's fastest Rubik's Cube solving robot, CubeStormer III[24], has kept almost all of its implementation a complete secret. In this project, we want to be able to contribute the following to the SpeedSolving community:

1. We present a reliable vision system for reading the state of the Rubik's Cube using Edge Detection, Adaptive Thresholding, pattern recognition, automatic white balancing and square prediction (section 4.5).
2. We detail a fast multithreaded implementation of Korf's algorithm using a perfect minimal hash function similar to a technique used in finding God's number [18] to save 75x more memory over a generic implementation and to speed up heuristic database lookups down to just  $\mathcal{O}(1)$ .
3. We present a new algorithm that is variation of Korf's algorithm using multithreaded Fringe Searching instead of IDA\* as well as an algorithm that combines Korf's and Kociemba's algorithm that can

outperform Kociemba's algorithm at shorter solution lengths and detail the shortcomings of Fringe Search within this particular use case.

4. We evaluate and compare the performance of different algorithms in terms of the speed that they can find a solution and the length of solution they give. In particular we compare, Korf's, Kociemba's and our various improvements and variations of Korf's algorithm.
5. We compare existing designs and explore grabbing mechanisms, gearing and motor controllers to demonstrate how accurate turning can be achieved using basic Lego Mindstorm kits. As well as evaluating our particular design to highlight the major hardware challenges in designing such a robot.

# Chapter 2

## Background

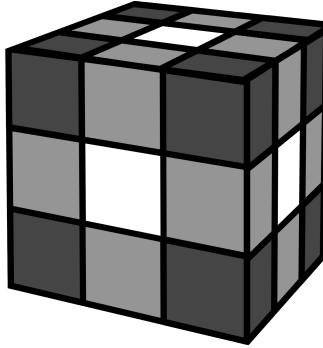
### 2.1 Fundamental Structure

The Rubik's Cube has many components to it and in order to obtain a model that we can reason about, we must have a consistent way of referencing distinct sections of the cube. This section details the most common basic notation.

#### 2.1.1 Rubik's Cube Jargon

1. A **face** refers to a single side of the cube comprised of 9 stickers.
2. A **cubie** refers to a smaller 'sub-cube' that builds up the bigger cube.
3. An **edge** is a type of cubie. It refers to the cubies that only have 2 colours attached to them. Figure 2.1 Light Grey.
4. A **corner** is another type of cubie. It refers to the cubies that have 3 colours attached to them. Figure 2.1 Dark Grey.
5. A **centre** is also a type of cubie. It refers to the cubies that only have 1 colour attached to them. Figure 2.1 White.
6. A **move** is the movement of a particular face by 90, 180 or 270 degrees.
7. A **rotation** is the movement of the whole cube without moving any faces.
8. A **facelet** refers to a sticker on a face.
9. A **speedsolver** refers to a person who attempts to solve the cube in the fastest time possible.

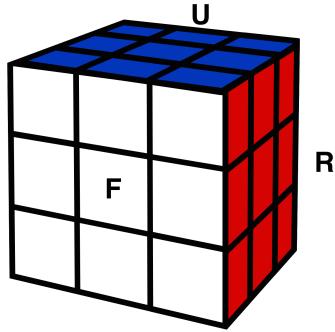
Figure 2.1: Labelled Cube



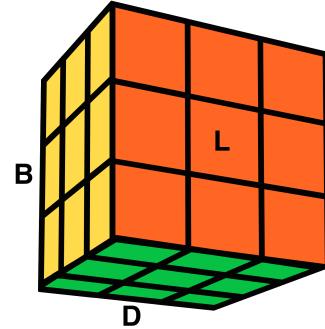
### 2.1.2 Face Notation

Usually one labels the faces of a Rubik's cube using the colour of its faces. E.g, For the official international colour scheme: Red, Blue, Yellow, etc. However, it is more useful to have a notation that is independent of face colour. This is because colour schemes vary from cube to cube. Instead we can label the cube using the direction that the face faces. Assume we have the official Rubik's cube with international colour scheme in a position such that the blue face faces upwards and the white face faces towards ourselves, we can label the faces as follows: F (Front), R (Right), L (Left), B (Back), U (Up) and D (Down)[5]. Figures 2.2a and 2.2b below shows this:

Figure 2.2: Face Notations



(a) Front side

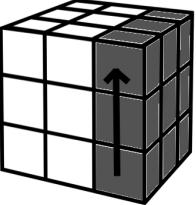
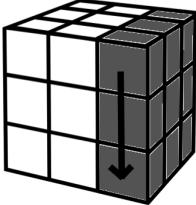
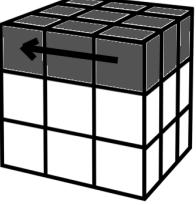
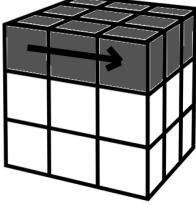
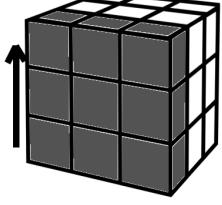
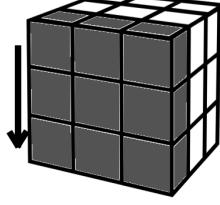
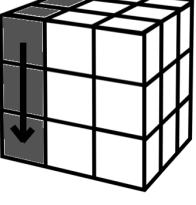
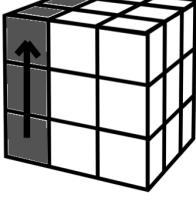
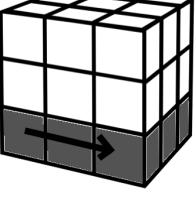
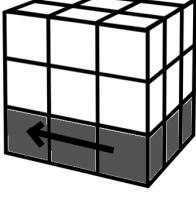
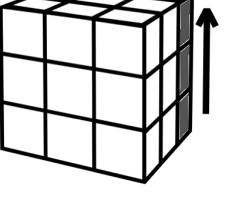
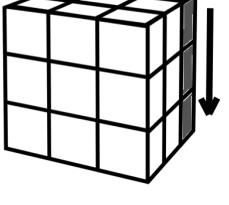


(b) Back side

### 2.1.3 Move Notation

Now that we've seen a notation for which we can refer to faces, we can now define a notation that defines moves that we can perform on the cube. We need two pieces of information to define a move: the face and the number of 90 degree turns clockwise[5]. For example: R1 is a 90 degree clockwise turn of the right face, L2 is a 180 degree turn of the left face and B3 is a 270 degree clockwise turn (or a 90 degree anticlockwise turn) of the back face. The table below shows all moves:

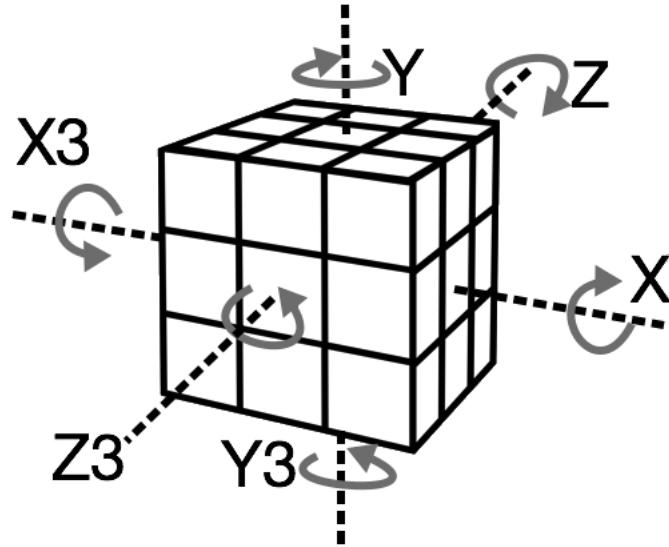
Figure 2.3: Move Notation Table

Move	Image	Move	Image
R		R3	
U		U3	
F		F3	
L		L3	
D		D3	
B		9 B3	

#### 2.1.4 Rotation Notation

So far, we have only defined which faces we can move. We can also express cube rotations[5] that rotate the whole cube. We can define how to rotate the entire cube by defining the axis of rotations X, Y and Z. If we draw a line through the R face to the L face as per figure 2.4, we define the clockwise rotation X as following the clockwise direction turn of the move R. Similarly, the Y clockwise rotation would follow the clockwise rotation of U in Figure 2.3 and Z clockwise rotation would follow the clockwise rotation of F.

Figure 2.4: Cube Rotations



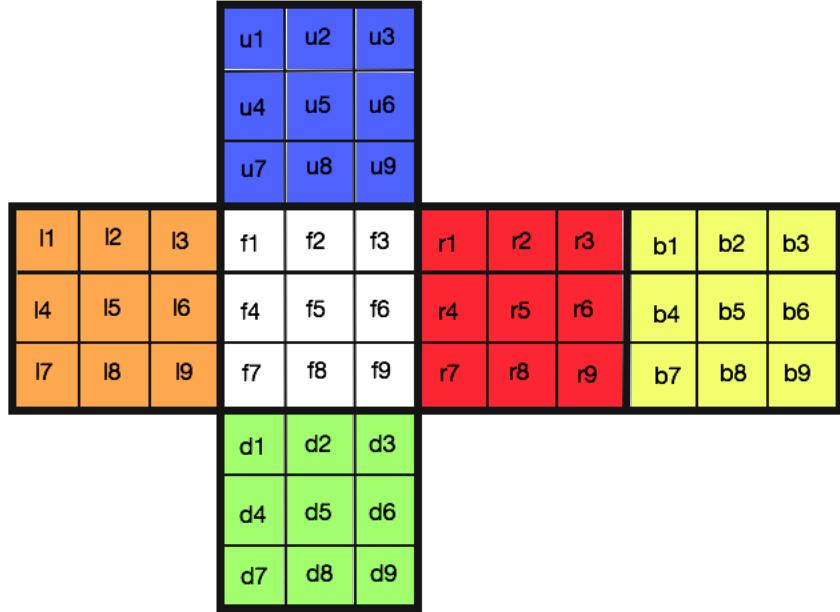
#### 2.1.5 Cube Notation

Now that we have face notation, we can introduce a notation to represent a cube state. If we imagine flattening out the cube into a net, we can label each of the 9 squares on each face as per Figure 2.5. Note, to prevent confusion with move notation, we will label each square with lowercase. E.g. r2 is not the same as R2. R2 refers to a 180 degree movement of the R face whereas r2 refers to the square r2. Since we only define moves where we move outer faces of the cube, the centres of each face remain fixed in their respective initial positions. This allows us to consistently map colours to faces. For example, if the blue centre piece is on the U face, all blue coloured squares could also be labelled as U.

We can now represent a cube state using a 54 character string. In the following order:

u1u2u3u4u5u6u7u8u9r1r2r3r4r5r6r7r8r9f1f2f3f4f5f6f7f8f9d1d2d3d4d5d6d7d8d9l1l2l3l4l5l6l7l8l9b1b2b3b4b5b6b7b8b9

Figure 2.5: Cube Net

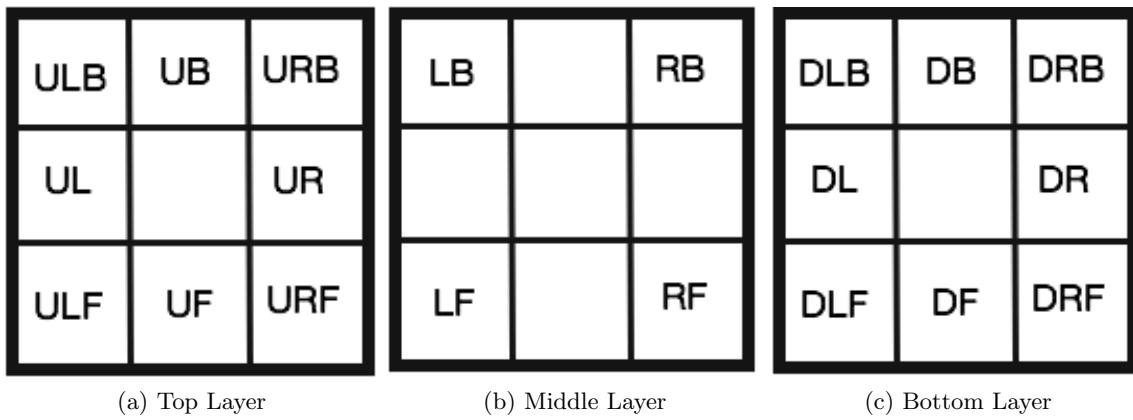


This notation is often useful for human input as we can just read the colours directly off the cube. It may sometimes be referred to as Facelet Level notation [13].

### 2.1.6 Cubie Notation

Sometimes it is easier to reason about the cube as being constructed of 3x3x3 cubies. Excluding the smaller cube that is directly in the centre, a cube consists of 26 total cubies: 12 *Edge* pieces, 8 *Corner* pieces and 6 *Centre* pieces. An edge piece can be uniquely identified using just the two faces that it touches. Similarly, a corner can be uniquely defined by the three faces it touches. Below shows a diagram where the cube is split into three layers by slicing twice through the XZ plane, creating a bottom, middle and top layer so that we can label all pieces:

Figure 2.6: Cubie Notation



### 2.1.7 Singmaster Notation

Another way of representing a cube is to use Singmaster Notation[21]. Singmaster Notation uses the piecewise notation and allows us to represent the cube in a much more compact form. The Singmaster Notation needs to account for two properties of a piece: its permutation (position in the cube) and its orientation (which way the piece is facing). Using the piece notation, we can define a cube as follows: UF UR UB UL DF DR DB DL FR FL BR BL UFR URB UBL ULF DRF DFL DLB DBR. That is, we put the actual piece that lies in each position UF, UR, UB, UL, etc. This determines the permutation. The orientation is determined by the order of how the piece is input. For example, if we defined a cube state as starting with UB FD..., this tell us that the piece that was in position UB in the solved state, is now in position UF. Similarly, the piece FD is in position UR. Notice the distinction between FD and DF: FD means that the F colour is facing the U direction and the D colour is facing the R direction. DF would mean a flipped version of this where the D colour is facing the U direction and the F colour is facing the R direction. This notation is often useful to reason about the Mathematics behind the number of states a Rubik's cube has.

## 2.2 The Mathematics

### 2.2.1 Laws & Lemmas

There are certain laws a Rubik's cube has which are often overlooked[11]. These properties are crucial to reach the true number of the size of the problem space, since it proves that some states are unreachable by using the moves defined in section 2.1.3 .

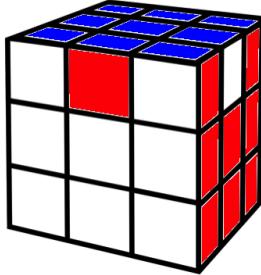
#### 2.2.1.1 All swaps are even

All reachable states are those which can be obtained using only an even number of swaps. A swap is defined as exchanging the position of a piece for another. This means the diagram in figure 2.7a shows an impossible state since the two pieces UF and UR have only performed one swap. A simple proof for this lemma is as follows: Using the legal moves defined in section 2.1.3 we can see that any move will always perform an even number of piece swaps. This means any combinations of moves will only perform an even number of swaps in total. Imagine a U move, this requires 4 edge swaps and 4 corner swaps. The same argument can be made for any other 90 degree move. Any 180 degree move, e.g. U2 requires 2 edge and 2 corner swaps.

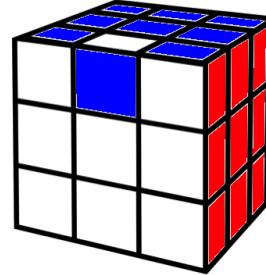
#### 2.2.1.2 All edge flips are even

Similar to section 2.2.1.1 above, all reachable states are those which can be obtained using only an even number of edge flips. For example, the state in Figure 2.7b below is unreachable since it requires only 1 edge flip. In order to reason about this, we must first define what a 'good' edge or a 'bad' edge is. A 'good' edge is an edge which can be permuted back to its original position and orientation using only moves involving faces U, R, D and L. A 'bad' edge would be an edge that cannot satisfy the 'good' edge condition. We can see that using only moves U, R, D and L from a solved cube state, all edges must be 'good' and can never turn 'bad'. This is because no matter how many moves we make, we can always recover the position and orientation of any edge piece by simply reversing the U, R, D or L moves performed. This means no number of U, R, D or L moves can flip an edge.

Figure 2.7: Illegal States



(a) Illegal state edge swap



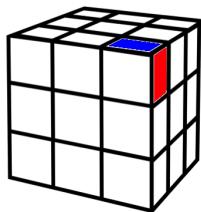
(b) Illegal state edge flip

With the remaining faces: F and B, any 90 degree move will flip all 4 edges on that face. We prove this by simply performing an F move and then attempting to flip any flipped edges on the F face using only U, R, D or L. We previously proved that U, R, D or L cannot flip edges so it will be impossible to return any edges on the F face to back to their original positions and orientations. Again, since we can only flip 4 edges at a time, the total number of edge flips for any reachable cube state must be even.

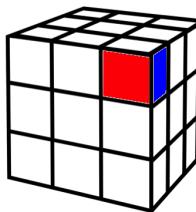
### 2.2.1.3 All corner orientation totals are divisible by 3

So far, we've only seen laws associated with edges. It is slightly harder to reason about corner orientations since there are 3 possible orientations per corner. Let us label the orientations by labelling the solved state as 0, solved state twisted clockwise as 1 and solved state twisted anti clockwise as 2. The diagram below shows all three corner orientation states and their labels:

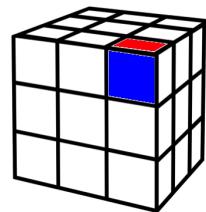
Figure 2.8: Corner states



(a) Corner 0



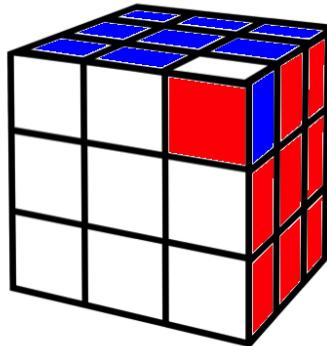
(b) Corner 1



(c) Corner 2

If we sum the labels of all corners of any reachable state of a cube, the total is divisible by 3. The diagram below shows a state that is not reachable since the sum of all the labels is 1.

Figure 2.9: Illegal state corner flip



Once again, we must define what it means to have a 'good' corner or a 'bad' corner. Notice that all corners lie on the U face or D face. This means for each corner, there is always a sticker that faces the U or D direction. A 'good' corner is defined as a corner where the sticker that faces the U or D face is of either U or D colour. Any other corner orientations are defined as 'bad'.

Using these definitions, we can see that any moves involving the U or D faces cannot change the orientation of any corners. For example, take solved cube and only perform U or D moves. All corners are 'good' in a solved cube state. No number of U or D moves can change these corners from 'good' to 'bad'. For the other R, F, L and B moves, a 90 degree turn will increment the orientation label of 2 corners by 1 and add 2 to the orientation label of another 2 corners (modulo 3). The total change is therefore  $1+1+2+2 = 6$  since each R, F, L, B will only add 6 to the total label sum. Since the total label sum of a solved cube is 0 and any move can the total change is can either be 6 or 0, the corner orientation totals must be divisible by 3.

## 2.2.2 Problem space

Now that we have an idea of the structure and laws of the Rubik's cube, we can now begin to reason about the problem space[20].

### 2.2.2.1 Orientations

We define the number orientation as the number of directions a piece can face towards.

Any edge piece can only have 2 orientations. This is obvious if we define some edge piece as XY, its other orientation is YX. There are no other possible orientations. Since there are 12 edges, we would think that the total number of edge orientations is  $2^{12}$ . However, using the edge lemma in section 2.2.1.2, we can reason that half of all edge orientations are unreachable since all odd numbered edge flips cannot be reached. This reduces the number of reachable edge orientations to only those with even edge flips:

$$\text{Edge\_Orientations} = 2^{12}/2 = 2^{11} = 2048 \quad (2.1)$$

Similarly, any corner piece can have 3 orientations. Since there are 8 corners, we would think that the total number of corner orientations is  $3^8$ . However, using the corner lemma in section 2.2.1.3, we can prove that only a third of all corner states are actually reachable since only those with a total corner label sum divisible by 3 can be reached. This reduces the number of corner orientations to:

$$\text{Corner\_Orientations} = 3^8/3 = 3^7 = 2187 \quad (2.2)$$

### 2.2.2.2 Permutations

We define the number of permutations as the number of positions a cubie/piece can be in.

Let us take any corner piece from the 8 corners. For any cube state, this corner piece can be in any 1 of 8 positions. We then choose a second corner piece. Since the first piece has already claimed a position, the second piece can only choose from 1 of 7 positions. This continues until the last piece. This gives us  $8 * 7 * 6 * 5 * 4 * 3 * 2 * 1$  possible corner permutations. The total number of corner permutations is:

$$\text{Corner\_Permutations} = 8! = 40320 \quad (2.3)$$

A similar argument can be made for edge permutations.

$$\text{Edge\_Permutations} = 12! = 479,001,600 \quad (2.4)$$

One may then argue that the total number of permutations is  $8!*12!$ . However, only half of these permutations are actually reachable using the legal moves defined in 2.1.3. Using our even swap lemma from section 2.2.1.1, we can reason that all states that have an odd number of swaps are not reachable which halves the number of permutations to  $12! * 8!/2$ .

### 2.2.2.3 Total state space

Using the values above, we can calculate the total size of the state space:

$$\text{Edge\_Size} = \text{Edge\_Orientations} * \text{Edge\_Permutations} \quad (2.5)$$

$$\text{Corner\_Size} = \text{Corner\_Orientations} * \text{Corner\_Permutations} \quad (2.6)$$

$$\text{Total\_Size} = (\text{Edge\_Size} * \text{Corner\_Size})/2 \quad (2.7)$$

This gives a total state size of 43,252,003,274,489,856,000.

## 2.2.3 Group Theory

### 2.2.3.1 What are groups?

A group is a structure which consists of a set and an operation that can combine any two elements[4]. There are four conditions called ‘group axioms’ that the set and operation combination must satisfy. Let  $G$  be our set and  $\overline{OP}$  be our operator:

1. **Closure** - Any two elements combined using the operator must give another element that is in the set. That is:  

$$\forall a, b \in G, \exists c \in G : a \overline{OP} b = c$$
2. **Associativity** - Order of evaluation does not matter. That is:  

$$\forall a, b, c \in G : a \overline{OP}(b \overline{OP} c) = (a \overline{OP} b) \overline{OP} c$$
3. **Identity** - The set must contain the identity element under the operation. That is:  

$$\exists a \in G, \forall b \in G : a \overline{OP} b = b$$
4. **Invertibility** There is an inverse element for every element in the set. That is:  

$$\forall a \in G, \exists b \in G : a \overline{OP} b = i$$
  
 Where  $i$  is the identity element defined previously.

An example of a group is the set of integers  $\mathbb{Z}$  and the operation  $+$ . It is easy to see how Associativity is satisfied. Closure is satisfied since the addition of any two integers will give another integer in  $\mathbb{Z}$ . The identity element is 0 since 0 added to anything will just give itself. Invertibility is satisfied because the inverse element of any integer  $i$  is  $-i$ .

### 2.2.3.2 How is this relevant to Rubik’s Cubes?

As it turns out, the set of all reachable Rubik’s Cube states with an operator that applies moves (let’s call this operator:  $*$ ) forms a group[4]. Any cube state can be expressed as a combination of move applications from the solved state. For example, let the solved state be  $C$ . We can say something like this:  $C * R * R = C * R2 = C * L * L3 * R * R$ . That is, applying two R moves gives the same state as applying R2 which also gives the same state as applying L, L3, R, R. You may notice that applying moves to a cube state should be an illegal operator since our operator should be combining cube states and not states and moves. However, we are combining 2 cube states. R is shorthand for  $C * R$ . So  $C * (C * R) * (C * R) = (C * R * R)$ .

Let us call the group that contains all reachable states and the move application operator:  $G_0$ .

Let us prove  $G_0$  is in fact a group by stepping through all the axioms[4]:

1. **Closure** - Since the group contains all the states reachable using legal moves and we can only apply legal moves using  $*$ , it is impossible to generate unreachable states and we therefore have closure.
  2. **Associativity** - Let us take any cube states  $S_1, S_2$  and  $S_3$ :  $(S_1 * S_2) * S_3 = S_1 * (S_2 * S_3)$ . We can see that this is the case since taking  $S_1$  and applying the sequence of moves that took  $C$  to  $S_2$  and then applying the sequence of moves that took  $C$  to  $S_3$  is exactly the same regardless of the evaluation order. For example, let’s use  $(C * R) * U = C * (R * U)$ . We can see that if we take a solved cube and apply the move R and then U is the same as taking a solved cube and then applying the cube state  $(R * U)$  which is just R followed by U.
  3. **Identity** - The identity is our solved state,  $C$ . Since  $C$  is the same as applying no moves.
  4. **Invertibility** - All reachable cube states must have an inverse. Let us take a cube state  $C * S$  where  $S$  is a sequence of moves. We can reverse any sequence by simply ‘undoing’ all the moves. For example, the cube state  $C * R3 * U * B$  has an inverse  $C * B3 * U3 * R$ .
- Proof:**
- $$\begin{aligned}
 & (C * R3 * U * B) * (C * B3 * U3 * R) \\
 &= (C * R3 * U * B * B3 * U3 * R) \text{ (by def. of identity } C \text{ and associativity)} \\
 &= (C * R3 * U * C * U3 * R) \text{ (by def. of } B \text{ inverse)} \\
 &= (C * R3 * U * U3 * R) \text{ (by def. of identity and associativity)}
 \end{aligned}$$

$$\begin{aligned}
&= (C * R3 * C * R)(\text{by def. of } U \text{ inverse}) \\
&= (C * R3 * R)(\text{by def. of identity and associativity}) \\
&= (C * C)(\text{by def. of } R \text{ inverse}) \\
&= C
\end{aligned}$$

### 2.2.3.3 Parity

We can define the parity of a permutation as whether the number of swaps required to obtain that permutation is even or odd[12]. An even permutation is a permutation that requires an even number of swaps. An odd permutation is one that requires an odd number of swap. Notice how this relates to the even swap lemma in section 2.2.1.1. Another way of expressing this lemma would be to say that the parity of all edge and corner permutations must be even.

## 2.2.4 Numbering Schemes

Numbering schemes become useful for reducing memory consumption. To see how we use these numbering schemes see section 4.1.2.2.

### 2.2.4.1 Factorial Numbering

Assume we have 4 numbers:  $\{0,1,2,3\}$ . There are  $4! = 24$  possible permutations. A factorial numbering gives us the ability to number each unique permutation with an integer between 0 and 23[25]. The easiest way to describe a factorial numbering scheme is to step through an example:

Let's assume we wish to number the permutation  $\{1,3,0,2\}$

- 0 is the first element in the original set of numbers. Now it lies in index **2**
- 1 is the second element in original set of numbers. Now it lies in index **0**
- 2 is the third element in the original set of numbers. Since the indexes 0 and 2 have already been taken, there are only two positions left that 2 could be in. If we label these positions as 0 and 1 from left to right then 2 lies in index **1**
- 3 is the fourth element in the original set of numbers. There is only one position left for it to be in so it lies in index **0**

From this, we see that our number is **2010**, but we aren't quite finished yet. This number is represented as a mixed radix number. We need to convert this to base 10.

Our factorial number can be expressed as follows  $2_4 0_3 1_2 0_1$  where  $x_b$  is  $x$  expressed in base b. To convert to base 10:

$$\text{base10}(2_4 0_3 1_2 0_1) = 2 * 4! + 0 * 3! + 1 * 2! + 0 * 1! = 50_{10}$$

### 2.2.4.2 nPr Numbering

Assume now that we have 6 numbers  $\{0,1,2,3,4,5\}$  but we can only use 4 of 6 numbers at any time. This gives us  ${}_6 P_4 = {}_6 C_4 * 4! = 360$  possible permutations. Again, we would like to label these permutations from 0 to 359[1]. For each digit we need to calculate:

$$\frac{(n - (i + 1))!}{(n - r)!} * \text{number of unused preceding digits} \quad (2.8)$$

Where  $i$  is the index of the number in the smaller set,  $n$  is the size of the set to choose from and  $r$  is the size of the smaller set and number of preceding digits are the number of digits preceding the current digit in the original set that have yet to be used. We then take the sum. The easiest way to describe this numbering scheme is to step through an example:

Let's assume  $n = 6$  and  $r = 4$  and we wish to number the permutation  $\{5,0,2,3\}$

- **5**,  $\frac{(6-1)!}{(6-4)!} * 5 = 300$
- **0**,  $\frac{(6-2)!}{(6-4)!} * 0 = 0$
- **2**,  $\frac{(6-3)!}{(6-4)!} * 1 = 3$ , since 0 has already been assigned
- **3**,  $\frac{(6-4)!}{(6-4)!} * 1 = 1$ , since 2 and 0 have already been assigned

Therefore the number for this permutation is  $300 + 0 + 3 + 1 = 304_{10}$

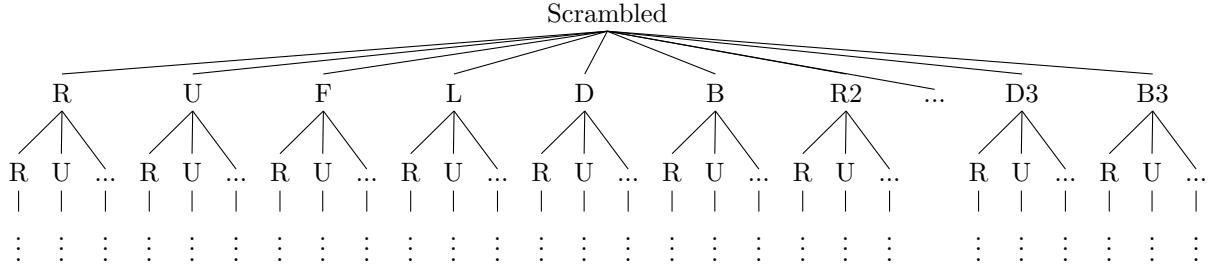
## 2.3 Existing Optimal Algorithms

This section aims to detail the existing algorithms for finding optimal solutions.

### 2.3.1 The Obvious Algorithm: Brute Force

The most obvious way to find a solution to a Rubik's cube is to just brute force search. That is, we can systematically (or randomly) explore all potential solutions until we find one. We can imagine our search space as a tree, using the diagram below:

Figure 2.10: Search Tree



#### 2.3.1.1 Breadth First Search

A breadth first search of a search tree aims to search each child of a node first before furthering the search to all grandchildren and great grand children...and so on. In the case of a Rubik's cube, we would try all 1 move solutions. Then all 2 move solutions and then 3...etc. In other words, let our scrambled cube state be  $S$ . We first try  $S*R$ ,  $S*U$ ,  $S*F$ ,  $S*L$ ,  $S*D$ , etc all the way until  $S*B3$ . If we do not find a solution, then we try  $S*R*R$ ,  $S*R*U$ ,  $S*R*F$ , etc. If we perform a breadth first search of this tree then we will eventually find an optimal solution. However, there are many problems with this approach:

- **Exponentially increasing size** - With each increasing depth level of our search tree, the branching factor of 18 means that the number of solutions we are required to look at will increase by a factor of 18. This will explode very quickly and become infeasible to search within a reasonable time.
- **Exponential increase in memory consumption** - A non-recursive implementation of a breadth first search requires a queue to store the child nodes to be explored. In the worst case, a 20 move solution would require searching  $18^{20}$  nodes. It is clearly not feasible to maintain a queue of this size.

To show the extent of how the number of nodes increase with depth up to 10:

Figure 2.11: Nodes generated at each depth

Solution Length	Nodes
1	18
2	324
3	5832
4	104,976
5	1,889,568
6	34,012,224
7	612,220,032
8	11,019,960,576
9	198,359,290,368
10	$3.5704672 * 10^{12}$

### 2.3.2 The First Real Attempt: Thistlethwaite's Algorithm

The first attempt at creating an optimal solution finder used group theory. Thistlethwaite's algorithm aims to break down the Rubik's cube into smaller sub-problems that can be calculated within a reasonable time[19]. The algorithm works by splitting the solve into 4 phases where we increasingly restrict certain moves. This will reduce the number of reachable states gradually until there is only one state left: the solved state.

Let us define the following groups:

$$G0 = \langle L, R, F, B, U, D \rangle \quad (2.9)$$

$$G1 = \langle L, R, F, B, U2, D2 \rangle \quad (2.10)$$

$$G2 = \langle L, R, F2, B2, U2, D2 \rangle \quad (2.11)$$

$$G3 = \langle L2, R2, F2, B2, U2, D2 \rangle \quad (2.12)$$

$$G4 = \{C\} \quad (2.13)$$

#### 2.3.2.1 Group G0

$G0$  is the group of all states reachable using moves  $L, R, F, B, U, D$ . Notice how this is just all reachable states using any of the legal moves defined in section 2.1.3 since we can perform any  $L2, R2, F2$ , etc moves by simply performing  $L * L$ ,  $R * R$ ,  $F * F$ , etc. Similarly we can perform any  $L3, R3, F3$ , etc by performing moves  $L * L * L$ ,  $R * R * R$ ,  $F * F * F$ , etc. Our aim is to move from  $G0 \rightarrow G1 \rightarrow G2 \rightarrow G3 \rightarrow G4$ . Where  $G4$  contains only the solved cube state.

#### 2.3.2.2 Group G1

$G1$  is the group of all states reachable using moves  $L, R, F, B, U2, D2$ . In contrast to group  $G0$ , the reachable states are smaller.  $G1$  contains only 'good' edges. To see why this is so, let us look back to our edge flip lemma in section 2.2.1.2. To explain why there are always an even number of flips, we proved that using only moves  $U, R, D$  and  $L$ , it is not possible to flip any edges. Instead of moves  $U, R, D$  and  $L$ , let us prove the same result is possible using moves  $L, R, F, B, U2$  and  $D2$ .

Let us perform an X rotation (described in section 2.1.4). Notice that if we rotate the cube, in order to rotate the same faces as in our previous orientation, our previous  $U$  moves would now be  $B$  moves,  $D$  moves would now be  $F$  moves and  $R$  and  $L$  moves would remain the same. This means that moves  $L, R, F$  and  $B$  also have the same property in that they cannot flip any edges. Now let's look at moves  $U2$  and  $D2$ . Since we've performed an X rotation, previous  $F$  moves are now  $U$  moves and previous  $B$  moves are now  $D$  moves. Remember in our edge flip lemma in section 2.2.1.2 we said that quarter turns of these faces would flip 4 edges. However, we don't have quarter turns. Instead, in  $G1$ , we only have  $U2$  and  $D2$  (180 degree turns). If we imagine these as 2 quarter turns, the first quarter turn would flip the 4 edges of that face. However, the next quarter turn would flip the same 4 edges back to their original orientations so these 180 degree moves cannot possibly flip any edges. This means that if we start from the solved state where all edges are 'good', all edges will remain 'good' assuming we only use moves  $L, R, F, B, U2, D2$ .

#### 2.3.2.3 Group G2

$G2$  is the group where we further restrict the reachable states to those reachable using moves  $L, R, F2, B2, U2, D2$ .  $G2$  contains only 'good' corners where 'good' corners are now defined as the corners which have an  $R$  or  $L$  sticker facing the  $R$  or  $L$  direction. To see why this is so, let us take moves  $L$  and  $R$ . None of these moves can change the orientation of the corners. Now let's look at  $F2, B2, U2$  and  $D2$ , none of these moves can change the 'good'-ness of a corner since they make any stickers facing left face right and any stickers facing right face left.

As well as only containing 'good' corners, we also fix edges in the centre layer in between the  $R$  and  $L$  faces. This means all edges that belong on the centre layer are on the centre layer but not necessarily

permuted correctly. To see why this is so, consider moves R and L. These cannot affect any edges in this middle layer. The remaining F2, B2, U2 and D2 moves can only change the permutation of the edges on the middle layer, it can never move them out.

#### 2.3.2.4 Group G3

G3 is the group where we restrict all quarter turn moves. G3 contains the states where the edges in the L and R faces are in their correct slices. Where slices are defined as the middle layer between any two opposite faces, the UD slice is the middle layer between U and D, the FB slice is the middle layer between F and B and the RL slice is the middle layer between faces R and L. G3 only contains edges in their correct slice because all 180 degree turns can only permute edges within their respective slices. In addition, G3 enforces that the parity of the edge permutations is made even (i.e. we need an even number of edge swaps). This is easy to see if we start from the solved state and only perform 180 degree turns - we only ever swap the positions of 2 edges. Using our even edge swap lemma in section 2.2.1.1, we can also say that the permutation of the corners is also even since the total number of swaps must be even in any cube state.

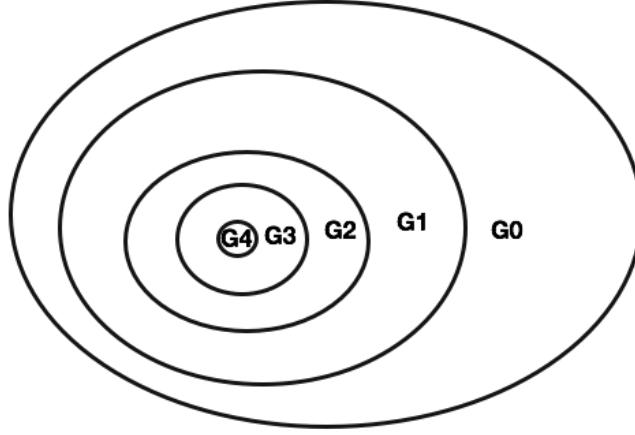


Figure 2.12: A visual representation of each group

#### 2.3.2.5 Pattern Databases

Most implementations of Thistlethwaite's algorithm use large pattern databases in order to quickly search for which moves are required for each transitioning phase. These pattern databases map substates of the cube to a sequence of moves that would take that state to the next phase. In this case, substates could be to only look at edge orientations for  $G0 \rightarrow G1$ . We would store which moves we would need for each possible edge orientation to move that orientation into a state where we would only have 'good' edges.

#### 2.3.2.6 What makes Thistlethwaite's algorithm so good?

This algorithm is effective because we reduce our search space to just searching for moves to transition between each group in a database. The size of each sub problem space is much smaller and therefore more manageable.

Below shows a table of the search space for each group transition:

Figure 2.13: Thistlethwaite's group transition size

Groups	Size
G0 → G1	2048
G1 → G2	1,082,565
G2 → G3	29,400
G3 → G4	663,552

### 2.3.2.7 What's so bad about it?

Although fast, this algorithm is not guaranteed to give an optimal solution. Below shows the worst case scenario in terms of number of moves to transition from one stage to another.

Figure 2.14: Thistlethwaite's group transition worst case

Groups	Worst Case
G0 → G1	7
G1 → G2	13
G2 → G3	15
G3 → G4	17

This gives a worst case scenario of 52 moves to solve a cube which is far from optimal.

### 2.3.3 A Different Approach: Korf's Algorithm

Korf's algorithm[15] takes a different approach to Thistlethwaite's algorithm. With Korf's algorithm, we move back to using the idea of brute force searching but we search a little more intelligently by using heuristics to prune branches from our search tree.

#### 2.3.3.1 Depth First Search

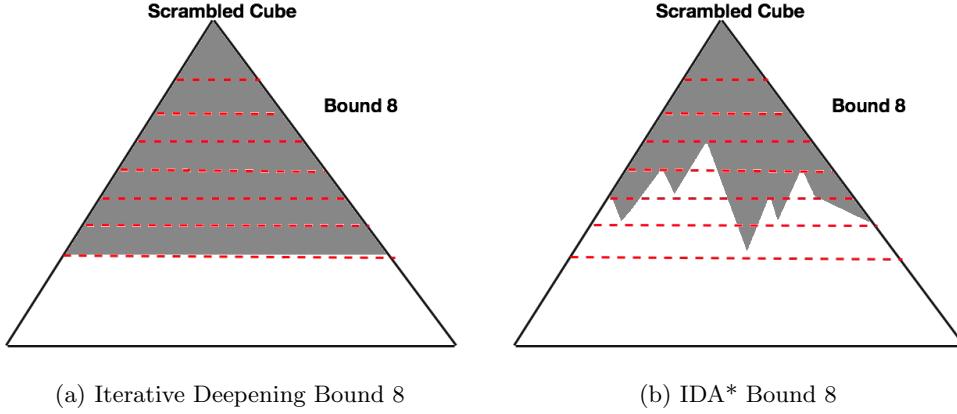
In contrast to Breadth First Search described in section 2.3.1.1, Depth First Search systematically searches the tree depth wise. That is, it explores the left-most child and then the left-most grandchild and then great grandchild... and so on until we hit a leaf node. For example, for a scrambled state  $S$ , we try  $S * R$ ,  $S * R * R$ ,  $S * R * R * R$ . In this case, since we know the maximum solution length is 20, we can stop going deeper after a solution of 20 is tried. So after  $(S * R)^{20}$ , we try  $(S * R)^{19} * U$ . The problem with depth first search is that it will not be guaranteed to find an optimal solution. This is because since we are sweeping the tree from left to right, we may well find longer length solutions first and terminate early when there may be a shorter solution later.

#### 2.3.3.2 IDA\* (Iterative deepening A\*)

The basis of Korf's algorithm is the IDA\* search algorithm. The IDA\* algorithm aims to reduce the search space by intelligently pruning branches that we know could never lead to a valid solution. It starts by searching for 1 move solutions using depth first search. If it finds nothing then it starts its search from the root again and tries 2 move solutions, then 3... and so on until it eventually finds a solution. This is the iterative deepening aspect of the algorithm. It solves the memory consumption problem that we had with breadth first search (since the most memory we would have to use is proportional to the depth of the tree, 20) whilst maintaining the requirement to find an optimal solution. This is because it always tries shorter solutions first.

The A\* part of the algorithm comes from the fact that it uses heuristics to estimate its ‘distance’ to our goal. In this case our goal is the solved state and the ‘distance’ is the number of moves required to solve a given state.

Figure 2.15: Iterative Deepening vs IDA\*



The highlighted areas in figure 2.15b and figure 2.15a represent what we actually search for a bound of 8. We can see that in IDA\*, we only search a subset of all possible states for a given bound since some branches have been pruned.

**2.3.3.2.1 Heuristics** The heuristic used must be admissible i.e it never overestimates the distance to the goal. Since there is currently no way to estimate the solution length for any arbitrary cube state, Korf's algorithm breaks down the heuristic into three smaller and easier to measure sub state estimates:

1. **Corners** - Number of moves to solve corners only
2. **6 of 12 Edges** - Number of moves to solve any 6 of the 12 edges only
3. **Remaining 6 of 12 Edges** - Number of moves to solve the remaining 6 of 12 edges only

The heuristic then combines all 3 estimates to form the heuristic  $h$ :

$$h(s) = \max(s_c, s_{e1}, s_{e2}) \quad (2.14)$$

where  $h(s)$  is the heuristic value of some state,  $s$ ,  $s_c$  is the number of moves needed to solved the corners,  $s_{e1}$  is the number of moves needed to solve the first 6 of 12 edges and  $s_{e2}$  is the number of moves needed to solve the rest of the edges. Since the maximum number of moves to solve any of these sub states would be less than or equal to the moves required to solve the whole cube, the heuristic is admissible.

**2.3.3.2.2 Using the heuristic** We first set up our search tree with the scrambled cube as the root node. We start searching the tree and fixing the bound to 1, i.e we try all 1 move solutions using a depth first search. We then try all 2 move solutions in a depth first fashion. When we wish to expand a cube state to explore its children, we use the heuristic measure to estimate the number of moves required to solve the cube from the given state.

Let the initial scrambled state be called  $m$ . Also, let the cube state we are questioning if we should expand be called  $n$ , we need two things to estimate the number of moves required to solve the scrambled cube. The first is the number of moves we have already executed to get from  $m$  to  $n$ , let's call this  $g(m, n)$ . The second is the estimate of the number of moves required to solve the cube from state  $n$ ,  $h(n)$ . We can now estimate the length of the solution that goes through node  $n$ ,  $f(m, n) = g(m, n) + h(n)$ . If the current bound of our search is  $b$ , we know that if  $f(m, n) > b$ , then there is no point in exploring any paths involving  $n$  since we are looking for a  $b$  move solution and so we can prune this branch.

As an example, let's assume we are currently searching for a 10 move solution. i.e. we've tried all 9,8,7,etc solutions and have found nothing. Let's now assume that we encounter a node  $n$  that we got to via 5 moves

from a scrambled state  $m$ . i.e.  $g(m, n) = 5$ . Additionally, let's estimate that from this point, we require 7 moves to solve. i.e.  $h(n) = 7$ . In this case, since we have a bound of 10, there is no point in expanding this node since our admissible heuristic told us that we would need at least 12 moves to solve it. We can prune this branch which significantly reduces the number of nodes we need to search.

### 2.3.3.3 What makes Korf's algorithm so good?

Although a standard depth first search is not guaranteed to find an optimal solution, Korf's algorithm is. This is because of the iterative deepening aspect of the algorithm. We first successively explore solutions of greater length until we find the first and shortest solution. E.g it is impossible to find a 10 move solution if the optimal solution is only 9 moves. This is because if we find a 10 move solution, that would have meant we explored all viable 9 move solutions first and found nothing which contradicts the fact that there is an optimal solution of length 9. Another advantage is that the IDA\* algorithm is very memory efficient. Since the maximum number of moves required for any solution is 20, the maximum stack size will also only be 20.

### 2.3.3.4 What's so bad?

Although Korf's reduces the search space, the average branching factor is still around 13[15]. This is still an exponential number of nodes to search with a worse case of around  $13^{20}$ . Korf's experimental results show that at a solution of depth 17 took around 2 days to finish and estimated that a depth of 18 would take around 18 weeks.

## 2.3.4 Improving Thistlethwaite's Algorithm: Kociemba's Algorithm

Kociemba's algorithm[14] improves upon Thistlethwaite's by reducing the number of phase transitions to just 2 instead of 4. This means we only need to transition between 3 groups:

$$G0 = \langle U, D, R, L, F, B \rangle \quad (2.15)$$

$$G1 = \langle U, D, R2, L2, F2, B2 \rangle \quad (2.16)$$

$$G2 = \{C\} \quad (2.17)$$

### 2.3.4.1 Group G0

The group G0 is the same as Thistlethwaite's algorithm. (All reachable states).

### 2.3.4.2 Group G1

The group G1 is equivalent to Thistlethwaite's algorithm's groups G2  $\langle L, R, F2, B2, U2, D2 \rangle$  but we've rotated the cube using a Z rotation which would make all previous L, R, F2, B2, U2, D2 moves into U, D, F2, B2, R2, L2 moves respectively.  $G0 \rightarrow G1$  is the same as Thistlethwaite's  $G0 \rightarrow G1 \rightarrow G2$ . Therefore the same properties hold: 'good' edges are always preserved, edges that belong on the UD-slice (layer between U and D faces) are now fixed but not necessarily permuted in their correct positions.

### 2.3.4.3 Group G2

The group G2 is just the solved state. To transition directly from  $G1 \rightarrow G2$  using only moves in G1 we must restore the permutations of all 8 corners. The 8 edges that lie on the U and D faces and the permutation of the 4 edges on the UD slice is the same as Thistlethwaite's  $G2 \rightarrow G3 \rightarrow G4$ .

### 2.3.4.4 The big difference

The major difference between Kociemba's and Thistlethwaite's stems from the consequence of merging together 4 phases into 2. Previously in Thistlethwaite's algorithm, we could generate pattern databases for moves to transition between each group. However, in Kociemba's, the transitions between each group are far too large. Instead, we have to perform smaller tree searches within each group in order to search for

a solution that will transition us from one group to another. The good news is that the maximum depths of these search trees are smaller than in Korf's. The maximum number of moves to transition between G0 and G1 is 12 and the number of moves to transition between G1 and G2 is 18. Most implementations of Kociemba's algorithm use IDA\* for this search.

#### 2.3.4.5 What's makes Kociemba's Algorithm so good?

Kociemba's algorithm is a good compromise between speed and length of solution. As mentioned in section 2.3.4.4, the maximum number of moves to transition between G0 and G1 is 12 and the maximum number of moves to transition between G1 and G2 is 18. This gives a maximum solution length of 30 which is not far from God's number: 20[18].

#### 2.3.4.6 What's so bad?

Although Kociemba's algorithm gives a close to optimal solution, we cannot guarantee that the solution is optimal. This is because when we search for solutions to transition from G0 to G1, we search for the shortest number of moves. Once we reach some state in G1, we look for the shortest number of moves from G1 to G2. Let us take an example where it takes us 11 moves to get from G0 to G1 and then 12 moves from G1 to G2. There may be a solution that costs 12 moves to get from G1 to G2 and then only 10 moves to get from G1 to G2 which gives an overall shorter number of moves. There are implementations of Kociemba's algorithm which continue to search for solutions so that we search the whole tree until we can prove that the solution we've found is actually optimal. However, these are exponentially slower and just like Korf's algorithm.

### 2.3.5 Why Not Human Algorithms?

So far, we've only looked at existing computing algorithms for find optimal solutions, but why not look at human algorithms? Most speedsolvers will not prioritise move count but instead turn speed. For a right-handed speedsolver, R and U moves are much easier to perform than other moves. Therefore, they tend to favour solutions that contain a lot of R and U moves. For a robot, this should only matter if the robot is restricted to perform R and U moves faster. Most human speedsolvers average around 50 - 60 moves which is far from optimal.

## 2.4 Existing Visioning Systems

Our aim is to somehow feed in data about the cube state so that we can begin to find a solution. There are some requirements we are looking to satisfy:

- **Fast** Must be able to quickly read the cube state colours
- **Reliable** Must be able to reliably read the colours of the cubes under certain lighting conditions
- **Robust** Must be able to read colours of the cubes under variable lighting conditions

### 2.4.1 Colour Schemes

In order to be able to read the colours of the cube state, we need to understand how colour can be represented[8].

**2.4.1.0.1 RGB Colour Scheme** All light is made from 3 component colours: Red, Green and Blue. We can create any colour using these 3 component colours by varying the intensities on each component. We can create white by having maximum intensity for all components, on the other hand, black can be made by having 0 intensity for all components. This is why any grey colour can be represented using a single intensity value.

#### 2.4.1.1 HSV Colour Scheme

HSV takes a different approach to representing colour. HSV uses three components: Hue, Saturation and Value. The hue determines the ‘wavelength’ of the colour within the visible light spectrum. In other words, it determines the ‘colour’ of the colour. Usually hue is represented as a colour wheel ranging from 0 to 360 degrees as show in figure 2.16 <sup>1</sup>.

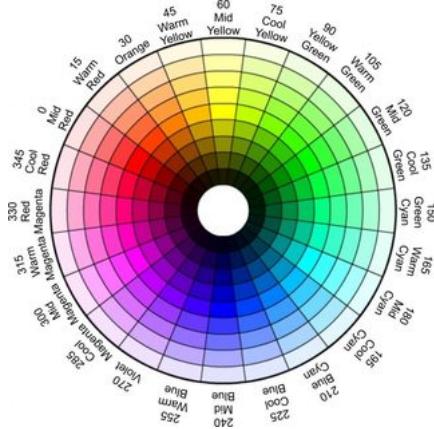


Figure 2.16: HSV Colour Wheel

Saturation determines the perceived intensity of the colour. It determines how ‘colourful’ the colour is. The closer the saturation is to 0, the more ‘dull’ it will look. A saturation of 0 will just give a grey image. Below shows an image of varying saturation for red<sup>2</sup>.

<sup>1</sup><http://i.imgur.com/PKjgfFXm.jpg>

<sup>2</sup><http://en.wikipedia.org/wiki/Colorfulness#/media/File:Saturationdemo.png>



Figure 2.17: HSV Saturation Demo

Value determines the brightness of the colour. The higher the value, the closer the colour will appear to white. The lower it is, the closer it will be to black.

## 2.4.2 Hardware

### 2.4.2.1 RGB sensors

The most primitive implementations of Rubik's Cube state readers use RGB sensors. The sensor will hover over each sticker of the cube in a pre-determined order to read the colour. As suggested by the name, the sensor reads the RGB values of the sticker, we can then manipulate this input to try to identify the colour.

Although simple, the major drawback with this approach is that it is difficult to distinguish between colours in varying lighting situations. E.g. if a cube is placed in a room with yellow light then the white may be mistaken for yellow. It is also very slow since we have to read one colour at a time. This is shown below<sup>3</sup>:



Figure 2.18: An example of an RGB sensor implementation

### 2.4.2.2 Camera

More advanced implementations of Rubik's Cube state readers use cameras to take pictures of each face. Attempts at this have previously been made and documented by Yakir Dahan and Iosef Felberbaum[6] in their CubeSolver<sup>4</sup> Android application in which the user is instructed to move the cube in front of the camera until their algorithms detect where the cube is and what the colours of each sticker are. There are other implementations of such a vision system out there and assumptions vary widely between them. Here are a few:

<sup>3</sup><http://imageshack.com/f/607/imag0130ql.jpg>

<sup>4</sup><https://play.google.com/store/apps/details?id=com.rubik.cubesolver>

- **Fixing cube position** - We assume the position of where the cube appears in the camera frame is fixed. We can then make assumptions about which coordinate a specific sticker of the cube would lie in. This is not reliable if the cube does not lie perfectly within the specified boundaries.
- **Fixed predictable lighting conditions** - We assume that the pictures taken of the cube are in predictable lighting conditions. This is so we can assume that colour values will always lie within specific boundaries. This is not robust if we take the cube into a different kind of lighting. E.g. if we assumed we would always have natural white light but we take the cube into a room with yellow light.
- **Fixed cube distance** - As the distance between the camera and cube increases, the cube will appear smaller and it will be harder to differentiate distinct squares. Similar to fixing the cube position within the camera frame, if the cube is too far away, then the vision system will be unreliable.

The sections below show potential ways to work around these assumptions.

### 2.4.3 Object tracking

In order to combat assumptions made about a fixed cube position in the camera frame and fixed distance assumptions between the cube and the camera, we can try to track where the Rubik's cube lies within the frame.

#### 2.4.3.1 Laplacian Operator

The Laplacian operator<sup>5</sup> can be used to detect the edges of an image[16]. If we are able to detect the edges of an object then we can begin to identify the object within the frame. Let us take a greyscale image. Edges in an image usually share a particular property: A major shift in intensity of the pixels around the edges. The more of a difference we have with neighbouring pixels, the more likely we are at an edge.

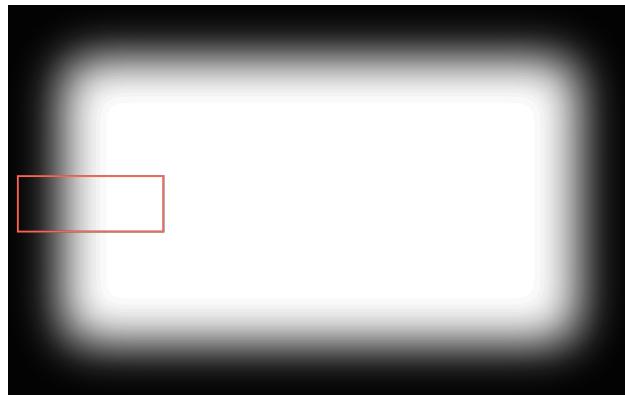


Figure 2.19: We wish to determine edges of this

Suppose we wish to detect the edges of the image above. Let us reduce this problem into a 1 dimensional problem and first plot the intensities of each pixel within the drawn square.

---

<sup>5</sup>All images and content adapted from: [http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/laplace\\_operator/laplace\\_operator.html](http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html)

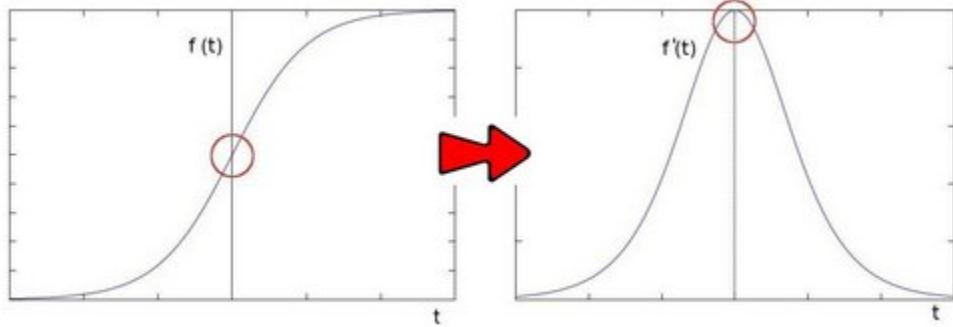


Figure 2.20: Getting the first derivative

The graph above on the left shows the plot of intensities of the pixels,  $f(t)$ . Let us take the first derivative of the graph. The graph in figure 2.20 shows the first derivative on the right,  $f'(t)$ , i.e. the change in pixel intensity. When the change in pixel intensity is at its highest (peak in the graph), we assume that this is an edge. So how do we find peaks in the graph? We know at the peak of a curve, the gradient is 0. So if we take the second derivative,  $f''(t)$  and look for values of  $t$  where  $f''(t) = 0$ , we can identify the peaks of  $f'(t)$ .

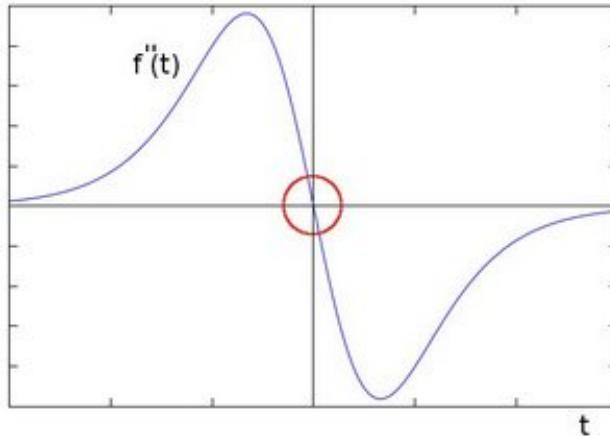


Figure 2.21: Second Derivative

More strictly, since we are working on a 2 dimensional image, in a 2 dimensional space, the laplacian operator is defined as:

$$\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \quad (2.18)$$

#### 2.4.4 Colour balancing

In order to combat the assumption made about predictable lighting, we can use colour balancing algorithms to neutralise any colour cast by coloured illumination on the cube. This will allow us to recognise colours independent of light source.

##### 2.4.4.1 Gray World Assumption

The Gray World Assumption[17] is a white balancing algorithm that assumes in a perfectly white balanced picture, the average colour is grey. That is, using the RGB colour scheme, the Red, Green and Blue values are all approximately equal. This essentially assumes that we have a good distribution of colours in the image.

**2.4.4.1.1 Estimation of illumination** This is the estimate of the colour casted by the incoming light. In its simplest form, Gray World Assumption computes the average of each colour channel of the image. Let us assume we have an  $N * M$  pixel image. Let's further assume that pixels are represented using the RGB colour scheme. The average of any given colour channel  $c$ , can be computed by:

$$\text{avg}_c = \frac{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \text{pixel}_c(m, n)}{M * N} \quad (2.19)$$

where  $\text{pixel}_c(m, n)$  is the colour channel  $c$  value of  $\text{pixel}(m, n)$

**2.4.4.1.2 Using the illumination estimate** Now that we have an average for each of the colour channels,  $\text{avg}_r, \text{avg}_g, \text{avg}_b$ , we must work out how much we need to normalise each pixel to make them a more neutral colour. Again, in its simplest form, the Gray World Assumption uses the average of all 3 channels to calculate the coefficient of adjustment for each channel. Let us name the coefficient of adjustment for channel  $c$  be  $S_c$

$$\text{avg} = (\text{avg}_r + \text{avg}_g + \text{avg}_b)/3 \quad (2.20)$$

$$S_c = \text{avg}/\text{avg}_c \quad (2.21)$$

We can now adjust each channel of each pixel:

Let  $\text{pixel}_{\text{orig}}$  be the original unadjusted pixel and  $\text{pixel}_{\text{balanced}}$  be the colour balanced pixel

$$\text{pixel}_{\text{orig}} = (R_{\text{orig}}, G_{\text{orig}}, B_{\text{orig}}) \quad (2.22)$$

$$\text{pixel}_{\text{balanced}} = (S_r * R_{\text{orig}}, S_g * G_{\text{orig}}, S_b * B_{\text{orig}}) \quad (2.23)$$

**2.4.4.1.3 Variations** The standard Gray World Assumption algorithm works well in most cases. There are variants that can improve the algorithm in some use cases.

**Normalising using max** A variant of normalisation is to use:

$$\text{max} = \max(\text{avg}_r, \text{avg}_g, \text{avg}_b); \quad (2.24)$$

$$S_c = \text{max}/\text{avg}_c \quad (2.25)$$

**Normalised Minkowski P-norms** Although in its simplest form, we use the average of each channel for our illumination estimate, another method of illumination estimate is to use p-norms in order to calculate  $\text{avg}_c$ . A p-norm as  $\text{avg}_c$  is defined as:

$$\text{avg}_c = \left( \frac{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \text{pixel}_c(m, n)^p}{M * N} \right)^{1/p} \quad (2.26)$$

Notice how the 1-norm just gives the average formula in equation 2.19

## 2.5 Existing Robots

There are currently a few Rubik's cube solvers out there. Each of them have various advantages and disadvantages to their designs.

### 2.5.1 MindCuber

#### 2.5.1.1 Design

MindCuber[2] is a single armed solver by David Gilday. The single arm is responsible for holding the cube in place whilst the lower platform rotates the D face of the cube. The single arm is also responsible for performing cube rotations. This robot can be built using an EV3 Lego Mindstorms set which gives it the advantage of being cheap to build. This can be seen in figure 2.22<sup>6</sup>. The limitations lie with its design. Since only a single side can be turned at a time, the cube needs to be rotated every time we wish to change the face we want to turn. This is extremely time consuming. The MindCuber uses a single RGB sensor to read each square individually.

**2.5.1.1.1 Algorithm** MindCuber is powered solely on the 'EV3 Intelligent Brick'. With only 64 MB of RAM and ARM-9 processor, the method used for solving the cube uses an undisclosed 'block-building' method which is far from optimal. Optimal algorithms require significantly more processing power and RAM in order to find a solution within a reasonable time.

Figure 2.22: MindCuber



### 2.5.2 JPBrown's CubeSolver

#### 2.5.2.1 Design

JPBrown's CubeSolver[3] was one of the first serious attempts to build a cube solving robot. The robot uses 3 arms built from Lego as shown in figure 2.23<sup>7</sup>. This allows it to move 3 independent faces without cube rotations. JPBrown's clamping mechanism uses a complex gearing system which makes the face move slowly but accurately. The vision system is webcam based. The cube must be presented in a very specific area of the camera frame. The robot itself is powered by 2 RCX Intelligent Bricks and a PC. The PC is responsible for finding a solution and parsing the camera frames for the vision. The Bricks are responsible for robot movement.

<sup>6</sup>[http://robotsquare.com/wp-content/uploads/2013/12/mindcub3r\\_s.jpg](http://robotsquare.com/wp-content/uploads/2013/12/mindcub3r_s.jpg)

<sup>7</sup><http://jpbrown.i8.com/cubesolver.html>

Figure 2.23: CubeSolver



**2.5.2.1.1 Algorithm** Since the CubeSolver has a PC at its disposal, Kociemba's algorithm was the algorithm of choice. This is because it will find a solution within a relatively short time and since the execution of moves is slow, the solution needs to be short.

### 2.5.3 Cubestormer

CubeStormer was developed by David Gilday and Mike Dobson[24]. CubeStormer uses 4 arms but not much else is known about its design since there is no official documentation. CubeStormer III took 18 months of development to improve on their previous Cubestormer II design. The robot is powered by an ARM CPU Smartphone and uses some variation of Kociemba's algorithm judging by the solutions it generates.

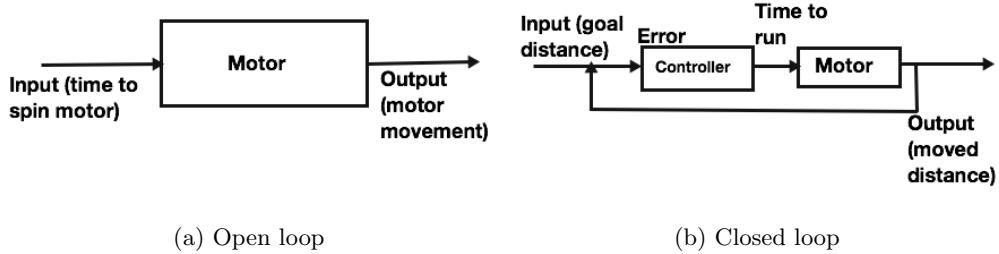
## 2.6 PID Controller

The PID (Proportional-Integral-Derivative)[7] controller is used in a closed feedback loop mechanism.

### 2.6.1 Open loop vs closed loop

What are open and closed loops? Let us take an example of a robot who wants to travel some distance X. An open loop approach would simply calculate how much time we want the motors of the robot to run based on its speed and the distance we want to travel. This is fine if the environment the robot runs in is always the same and the motors always spin at the same speed, but what do we do if the environment changes? For example, some sand gets stuck in the robot's motor and slows it down. The robot would stop short of our desired distance X. An open loop is not very effective for accurate and repeatable movements and therefore not very ideal for turning a Rubik's Cube face. We need the movements to be both repeatable and accurate. This is where the closed loop comes in. A closed loop uses a feedback mechanism in order to give the robot information about how its motors are spinning. This way the robot can adjust its movements based on the feedback.

Figure 2.24: Open vs Closed Feedback loop



We can see in the diagram above, the difference between an open and closed loop. The open loop gets a single input and then blindly attempts reach a goal given this piece of information. On the other hand, a closed loop has a few stages:

1. Input goal distance X
2. Calculate difference between current distance and goal distance X. (Error)
3. The controller then inputs the time to run to the motor. If the motor isn't as far as it expected to be, then we can tell it to run for slightly longer. Likewise, if the motor is closer than expected, we can tell it to run shorter.
4. The motor then outputs the distance it has moved back to the controller so it can repeat the same thing again.

This continues until the robot reaches its goal.

### 2.6.2 What is PID specifically?

There are many controllers but the PID controller is the most popular controller for its simplicity and robustness. The PID controller is defined as:

$$PID\ Controller = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \quad (2.27)$$

where  $t$  is time and  $e(t)$  is the error term at time  $t$ .

The PID controller is constructed from 3 terms:

- $K_p e(t)$  This proportional term determines how much we scale up or down the error term by adjusting the gain,  $K_p$ . A higher gain makes the system more sensitive and responsive to change. If the gain is too high, the system can become unstable and oscillate and never reach the goal. If the gain is too low, the system may be unresponsive to small errors.
- $K_i \int_0^t e(\tau) d\tau$  This integral term determines the accumulated offset up until time  $t$ . In effect, if we were to plot our error against time, it is summing the area under the curve until time  $t$ . Imagine a case where we use a motor on a sticky floor. As we approach the goal distance, we want to reduce the amount of power we give to the motor to make sure we don't overshoot the goal. If the power becomes too low, we end up standing still and not moving towards the goal, the sum of errors becomes increasingly large and this would then give more power to the motors. We adjust this term by scaling  $K_i$ . A high value will accelerate the error towards 0. Too high and it may overshoot.
- $K_d \frac{d}{dt} e(t)$  This derivative term calculates the change in error over time. It is scaled using  $K_d$ . This term helps to improve stability and can reduce the settling time.

We tune each of the K constants for different use cases.

### 2.6.3 What's so great about PID?

PID controllers give very accurate movement because they dampen the motor speed as we approach the goal distance. Not only does this give us a lower chance of overshooting the goal distance, it also means we have time to make minute adjustments as we approach the goal. This makes it fantastic for applications which require a high degree of accuracy.

# Chapter 3

## Design

In this chapter we detail and discuss the design of the system.

### 3.1 Overall Design

The system can be broken down into three components:

1. **Algorithm** - The algorithm used to find a solution to the Rubik's cube.
2. **Vision** - The method used in reading the state of the cube.
3. **The Robot** - The robot used to physically solve the cube.

#### 3.1.1 Algorithm Design

The search algorithm we will use to find a solution for the Rubik's cube will be run from a PC. We chose to use a PC so that we have more RAM and CPU power to find better solutions. The algorithm will be a mixture of Korf's and Kociemba's algorithm. Since Korf's algorithm does not guarantee that it can find a solution within a reasonable amount of time, a key aspect of this project will be to attempt to speed up Korf's algorithm as much as we can. Kociemba's algorithm will be used in cases where it is not possible for Korf's algorithm to return within a reasonable time (a 'fall-back' if you will). To keep the coding language consistent with the vision aspect of the system, Java will be the language of choice. Additional advantages include: simple networking API to allow the Android Smartphone to communicate with the PC, platform independence and garbage collection. Garbage collection is particularly useful for complex algorithms that generate many search nodes. With so many aspects to the project, any way to simplify the implementation will be welcome.

##### 3.1.1.1 2s Notation vs 1s notation

There are 2 main types of move notation we must consider for our search algorithm. The first, 2s notation, allows 180 degree movements to be counted as 1 move. This gives 18 possible one move moves in total: R, R2, R3, U, U2, U3, F, F2, F3, L, L2, L3, D, D2, D3, B, B2, B3. The second, 1s notation, only allows 90 degree turns. This gives 12 possible one move moves in total: R, R3, U, U3, F, F3, L, L3, D, D3, B, B3. It has been proven that the maximum number of moves required to solve any state using 2s notation is 20 and the maximum number of moves required to solve any state using 1s notation is 26[18].

**3.1.1.1.1 Which one is better for searching?** Although initially it may seem that 1s notation should be favoured since it explodes less quickly than 2s notation (12 vs 18), if we compare them when searching for long solutions, we will see that 2s notation produces less nodes: 1s notation has  $12^{26} \approx 1.14 * 10^{28}$  nodes for a 26 length solution whereas 2s notation has only  $18^{20} \approx 1.27 * 10^{25}$  nodes for a 20 length solution - 1000 times less! 1s notation may find short solutions more quickly since it explodes less quickly, but in reality, most solutions will require at least 10 moves in 2s notation, which definitely gives 2s notation the edge.

### 3.1.2 Vision Design

The aim of the vision system is to be able to detect the position of the cube within the camera frame regardless of where the cube is placed and independent of background, as well as have robust colour detection. This will be coded in the form of an Android application in Java to make use of an Android Smartphone camera.

In order to accomplish this, we will use a well known Open Source Computer Vision (OpenCV) library. OpenCV provides a myriad of tried and tested Computer Vision methods such as Canny Edge Detection, Laplacian Operators, Gaussian Blur, etc. The OpenCV library is available for Android which is perfect as we are using an Android Smartphone. This will allow us to quickly build an app that can scan all the sides of the cube without having to worry about the deep technical details of individual Computer Vision methods. We can use Android's built in video API and OpenCV's video frame processing libraries in order to achieve most of what we want.

### 3.1.3 The Robot Design

We decided on a Lego MindStorms robot for a number of reasons:

- **Availability** - The Department of Computing Robotics department already has several Mindstorm kits. This means we can build the robot without having to worry about resource limitations.
- **Flexibility** - A Lego system gives us flexibility. That is, we can build almost any design without limitation.

There were a number of options available for the actual robot design. On one hand, a 1 armed robot would be fairly simple to implement and cheap to build. Since we had so many sets of Lego Mindstorms at our disposal, we could afford to have more than a single arm. A four armed robot was chosen over a three armed robot for a number of reasons including:

- **Stability** - Having more arms allows us to grip the cube more easily, allowing for a more stable design.
- **Speed** - Having more arms means we don't have to rotate the whole cube as many times during a solve. Less moves means fast solve times.
- **Simple** - Less cube rotations also reduces the complexity of the solve.

We chose to have 4 arms that grasp onto faces R, L, B and F as it means our robot can lie horizontally flat on the ground which is a stable structure.

#### 3.1.3.1 NXT Servo Motors

NXT Servo Motors give us the ability to measure speed and distance. They have built-in rotation sensors that allow for motor movement accurate to  $\pm 1$  degree. The motors also have a series of gears inside of the housing so they are capable of producing a lot of torque. This can be seen in figure 3.1 <sup>1</sup>.

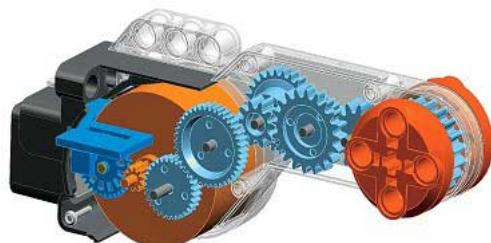


Figure 3.1: The insides of a servo motor

---

<sup>1</sup>[http://www.philohome.com/nxtmotor/motor1\\_3.jpg](http://www.philohome.com/nxtmotor/motor1_3.jpg)

### 3.1.3.2 NXT Intelligent Brick vs BrickPi

A major choice we had to make was whether we wanted to use the classic NXT Intelligent Brick or BrickPi<sup>2</sup>. The BrickPi offers many advantages over the NXT. The BrickPi runs on top of a Raspberry Pi and along with this gives us a Linux programming environment, WiFi capabilities and capacity to control more motors. Naturally, BrickPi was our first choice.

There was a major drawback, however. We found the motor controllers on the BrickPi board were very temperamental. The torque needed to turn the cube face would often burn out the motor controller. Numerous attempts to change design and power parameters were unsuccessful and would slow down the arm movements.

Another major problem we found was that when we tried to power 4 motors simultaneously, the voltage supplied with a DC 12v battery would not be sufficient to power all 4 motors and the Raspberry Pi. The Raspberry Pi would often reboot when the voltage would drop below its operating threshold. Halfway through the project we decided to make the switch to NXT instead. The NXT motor controllers are far more robust than those of their BrickPi counterparts, and we did not suffer from any power issues. The drawback is that we are only able to control 3 motors at most with each Brick and we are forced to use Bluetooth instead of WiFi. The NXT Bluetooth interface can only have 1 inbound connection and 3 outbound connections which is sufficient for a master-slave configuration but does not give us any room to change into a more complex configuration.

### 3.1.3.3 Lejos

Lejos<sup>3</sup> is an NXT firmware that gives us a Java programming environment. This makes the programming language consistent with the rest of the system. It also offers a well documented Robotics API which is great for those who have never tried Lego programming before. The Lejos libraries also give us a lot of other useful functionality such as an automatic closed feedback PID controller to keep motors in their positions after moving. This is particularly useful for holding the Rubik's cube in place. The PID controller also gives us far more accurate movement than BrickPi's basic API. We can see why this is so in our PID controller explanation in section 2.6.

The BrickPi API offers a very simple motor controller. In essence, it works as follows: We move our motor for 100ms at a constant speed, and then check if we've reached our goal distance. If not, we repeat the motor movement for another 100ms. This continues in a loop until we hit the goal distance. Since there is no dampening towards the goal, the BrickPi motors are very likely to overshoot the goal distance. A disaster for this system because we need very accurate movements. An error of millimetres off a perfect 90 degree turn could cause chaos!

### 3.1.4 Summary of Design

The aim of the system will be to do the following:

1. Take pictures of each side of the cube using the Smartphone's camera
2. Build the cube state
3. Send the cube state from the Android Smartphone to the PC via WiFi
4. Compute the solution to the given cube state on the PC
5. Send the solution from the PC back to the Android Smartphone
6. Use the Smartphone to control each of the 4 arms of the robot to solve the cube

---

<sup>2</sup><http://www.dexterindustries.com/BrickPi/>

<sup>3</sup>We specifically used NXJ version 0.9.1beta-3 from: <http://www.lejos.org/nxj.php>

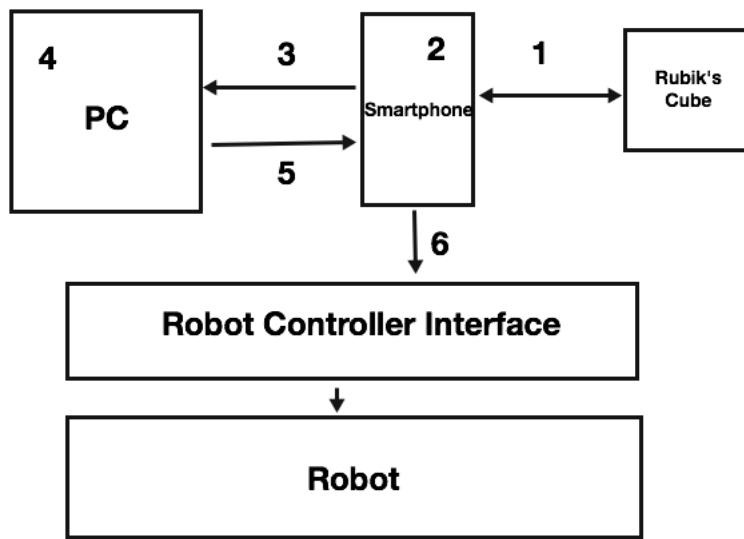


Figure 3.2: Overview of system through each stage

# Chapter 4

# Implementation

## 4.1 Korf's Algorithm

We have already seen in section 2.5 that most systems with enough RAM and CPU power will use Kociemba's algorithm for its speed to find a sub-optimal solution. Korf's is rarely ever used even with the RAM and CPU power, since it still has a branching factor of about 13, meaning it will not terminate within a reasonable time. In reality, we have 15 seconds to search for solution (Official SpeedSolving Rules<sup>1</sup>) before the stopwatch starts. So how far can we search in Korf's algorithm within these 15 seconds? This chapter details our implementation of Korf's algorithm and attempts to speed up searching.

### 4.1.1 Cube representation

Our cube representation attempts to compact the cube into as small a representation as possible[18]. When generating our heuristic pattern databases, we will need to store millions of cube states at once so a saving of a few bytes now could snowball into larger savings later. Remember from section 2.1.7 that a cube can be represented by where each of the corners and edges lie using Singmaster notation. The problem with Singmaster notation is that it requires a 48 character String (3 chars for each corner and 2 chars for each edge). In Java, we can estimate the amount of memory that this String would take up. If we assume each character takes up 2 bytes then each cube representation is 96 bytes! This is pretty expensive: around 10 million states would cost close to 1GB of memory. We can do better than this, so let's see how small we can go.

#### 4.1.1.1 Corners

Since there are 8 corners, we can represent the position of any of the 8 corners by an integer from 0 - 7 which can easily be represented by 3 bits per corner, giving a total of a 24 bits. However, we still need to handle orientation of the corner. Since a corner can only have three states, we can represent this with an integer between 0-2 which can be represented in 2 bits. This means any corner can be represented in 5 bits. Since the smallest amount of declarable memory in Java is a single byte, we can represent any corner using a single byte. This gives us an 8 byte representation of the state of all corners. In our implementation we store an array of 8 bytes. Below shows a table of how we've chosen to label our corners:

---

<sup>1</sup>Rule A3a1: <https://www.worldcubeassociation.org/regulations/#article-10-solved-state>

Figure 4.1: Corner labelling

Corner	Label	Corner	Label	Corner	Label
UBL	00000000	LUB	00001000	BLU	00010000
URB	00000001	BUR	00001001	RBU	00010001
ULF	00000010	FLU	00001010	LFU	00010010
UFR	00000011	RUF	00001011	FRU	00010011
DLB	00000100	BDL	00001100	LBD	00010100
DBR	00000101	RDB	00001101	BRD	00010101
DFL	00000110	LDF	00001110	FLD	00010110
DRF	00000111	FDR	00001111	RFD	00010111

The byte in an element of the array gives the orientation of the corner in the first 2 least significant bits and the remaining 3 bits gives the position of where that corner lies on the cube.

#### 4.1.1.2 Edge

Edges are expressed analogously to corners but since there are 12 edges, we need 4 bits to represent an edge's position. Edges can only have 2 orientations so only a single bit is required. Just like corners, we can represent any edge with just 5 bits. Again, since Java's smallest unit of declarable memory is a single byte, we require 12 bytes to represent all edges. In our implementation we store an array of 12 bytes. Below shows a table of how we've chosen to label our edges:

Figure 4.2: Edge labelling

Edge	Label	Edge	Label	Edge	Label
UB	00000000	LB	00001000	DB	00010000
BU	00000001	BL	00001001	BD	00010001
UL	00000010	RB	00001010	DL	00010010
LU	00000011	BR	00001011	LD	00010011
UR	00000100	LF	00001100	DR	00010100
RU	00000101	FL	00001101	RD	00010101
UF	00000110	RF	00001110	DF	00010110
FU	00000111	FR	00001111	FD	00010111

The byte in an element of the array gives the position of where that edge lies in the first 4 least significant bits and orientation in the final bit.

#### 4.1.1.3 Pulling it all together

We can now express any cube by combining our Edge and Corner arrays. This gives a total size of just 20 bytes for any cube state - almost 5 times smaller than Singmaster Notation! This representation also brings many other advantages:

- **Faster reads** - We can obtain any edge or corner status with just a single read in the compact form. Reading a piece's state from Singmaster Notation, we would need at least 2 reads (for an edge).
- **Fast writes** - If we wish to change the state of the cube, we can just write to an array with a single write. Since Strings are immutable in Java, an edit to the cube state in Singmaster notation would require us to build a new object each time we wrote. This is wasteful and slow.

#### 4.1.1.4 Move transition

Being able to move the cube is one of the most important operations. This operation will be happening billions if not trillions of times per search. That is why it is important to have this operation be as fast as

possible. In order to move as quickly as possible, we precompute move tables that tell us how a corner's or edge's position and orientation will change given a move. No computation is needed during the search since we've already predetermined where each piece will go. This will allow us to move with just 20 reads and 20 writes:

Figure 4.3: Move method

```

1 public class CompactCube{
2     ...
3
4     public void move(int move){
5         corners[0] = cornerTransitions[move][corners[0]];
6         corners[1] = cornerTransitions[move][corners[1]];
7         corners[2] = cornerTransitions[move][corners[2]];
8         ...
9
10        edges[0] = edgeTransitions[move][edges[0]];
11        edges[1] = edgeTransitions[move][edges[1]];
12        edges[2] = edgeTransitions[move][edges[2]];
13        edges[3] = edgeTransitions[move][edges[3]];
14        ...
15    }
16    ...
17 }
```

You may notice that this method could have easily been written as two `for` loops. The reason why we chose not to use a loop is because a loop would translate to jumps in the code. This is not necessary for something so simple and the sequential code will be much faster to execute. In this case, we sacrificed code size for a little bit of extra speed. The same technique was used in the God's number experiment[18].

#### 4.1.2 Heuristic generation

Now that we have a representation of the cube that we can manipulate, we can begin to generate our heuristic pattern databases. Remember that Korf's algorithm needs three pattern databases for its heuristic:

- Least number of moves to solve any corner states. There are  $8! * 3^7 = 88,179,840$  possible corner states.
- Least number of moves to solve 6 of 12 edges. There are  ${}_{12}P_6 * 2^6 = 42,577,920$  possible 6 edge states.
- Least number of moves to solve the remaining 6 edges. There are  ${}_{12}P_6 * 2^6 = 42,577,920$  possible 6 remaining edge states.

##### 4.1.2.1 Heuristic storage: First Attempt

Heuristic pattern database lookups need to be fast. The heuristic look ups will happen as many times as nodes are generated. A possible way to do this is to use a `HashMap`. In the best case, a `HashMap` will have a  $O(1)$  lookup, the worst case will have a  $O(n)$  lookup. However, there are a couple of problems:

- **Memory size** - Java `HashMaps` have a large memory overhead for each entry in the table. The overheads will become excessive once we have 88 million entries.
- **File size** - When it comes to storing the database on file, we will need to store up to 88 million hashed corner states and the number of moves required to solve them.

`HashMaps` in Java generally dynamically change their capacity depending on the load factor given. The load factor is a number between 0 and 1 which determines how full the `HashMap` can be before we have to expand it. This is to maintain its  $O(1)$  lookup time. Each entry in the `HashMap` uses  $32 * \text{Size}$  bytes. As

well as this, it uses  $4 * Capacity$  for each entry array (for when collisions occur). The default load factor is 0.75. We can use this to estimate the size of a `HashMap` implementation in memory[22]:

$$HashMap\_Mem\_Consumption \approx 4 * Capacity + 32 * Size\ Bytes \quad (4.1)$$

We can estimate the capacity by using  $0.75 * Size$ . For our corners table, this is:  $4 * 0.75 * 88179840 + 32 * 88179840 \approx \mathbf{2.9GB}$ ! Likewise, both edge tables would take  $2 * (4 * 0.75 * 42577920 + 32 * 42577920) \approx \mathbf{2.8GB}$ . That's almost **6GB** of memory on heuristic pattern databases alone! We can do better.

#### 4.1.2.2 Minimal Perfect Hash Function: Second Attempt

A perfect hash function one that can take a key set of size  $N$  and map it to a set of integers of size  $N$  with no collisions. A **minimal** perfect hash function is one that can map a key set of size  $N$  and map it to a set of  $N$  sequential integers.

**4.1.2.2.1 Corners** Ignoring orientations, corners can have  $8!$  permutations. Since we've labelled our corners from 0 - 7 (ignoring orientations), we can use the Factorial Numbering Scheme in section 2.2.4.1 to number each permutation of our corners.

We can then number the number of corner orientations[25]. There are  $3^8$  possible orientations but the orientation of 7 corners automatically gives the orientation of the last corner. Using the corner lemma mentioned in section 2.2.1.3, we can prove that there are actually only  $3^7$  orientations. Since each orientation is a number between 0 and 2, we can concatenate 7 of 8 corner orientations which gives a number in base 3. Since each orientation in base 3 is unique, when converted to base 10, it will give a unique numbering for each orientation.

Now we can combine the permutation number and orientation number using a cartesian product counting scheme:

$$\text{minimal\_hash}(\text{cornerState}) = \text{perm\_number} * 3^7 + \text{orientation\_number} \quad (4.2)$$

**4.1.2.2.2 Edges** Ignoring orientations, edges have  ${}_{12}P_6 = 655280$  permutations. We can use the nPr numbering scheme described in section 2.2.4.2 to give each permutation a unique numbering.

There are  $2^6$  possible orientations for 6 of 12 edges. In this case, since every orientation is a number between 0 and 1, we can concatenate 6 of 12 edge orientations to give a binary value. We can convert this to base 10 to give a unique mapping.

We can now combine the permutation and orientation number using a cartesian product counting scheme analogous to corners:

$$\text{minimal\_hash}(\text{EdgeState}) = \text{perm\_number} * 2^6 + \text{orientation\_number} \quad (4.3)$$

**4.1.2.2.3 Faster Heuristic Lookup** Now we have a way of encoding corner and edge states into sequential integers! We can now store a byte array where the index is the encoded state. Indexing by index  $i$  will give us the minimum number of moves required to solve state  $i$ . A similar technique was used in finding God's Number[18].

**4.1.2.2.4 NibbleArray** We can still do better! After the first set of pattern database generation, we found that the maximum minimal number of turns required to solve any corner state is 11 and the maximum minimal number of moves to solve any 6 of 12 edge states is 10. This means we need just 4 bits to represent each state's move count. Since Java's minimum size for memory declaration is a single byte, we decided to write a custom `NibbleArray` class that allows us to store 4 bits. Even for our largest table of 88,179,840 corner states we only use  $4 * 88,179,840\ bits \approx \mathbf{42MB}$ . For both edge arrays we only use  $4 * 2 * 42,577,920 \approx \mathbf{40MB}$  giving us a memory consumption total of **82MB**. That's a massive 75x smaller than the `HashMap` implementation!

**4.1.2.2.5 Using the Heuristics** So how can we generate these tables? In our implementation, we've used a work queue approach. Let's use the corner heuristic generation as an example. Figure 4.4 shows some Pseudo Java-like code for corner generation:

Figure 4.4: Main Corner generation loop

```

1  while(!workQueue.isEmpty()){
2      cornerState = workQueue.pop();
3      moveCount = cornerStates[encodeCorners(state)];
4
5      for(int move = 0; move < NUMMOVES; move++){
6          //Move corners
7          newCornerState = moveCorners(move, cornerState);
8
9          //The new corner encoding
10         cornerEncoding = encodeCorners(newCornerState);
11
12         if((cornerStates[cornerEncoding] == 0 ||
13             cornerStates[cornerEncoding] >
14             (moveCount + moveCost[move])) &&
15             cornerEncoding != 0){
16
17             workQueue.add(newCornerState);
18             cornerStates[cornerEncoding] =
19                 (moveCount + moveCost[move]));
20         }
21     }
22 }
```

The work queue starts with the solved cube state. On each iteration, we take a corner state from the workqueue and find its corner encoding described by the perfect minimal hash function in section 4.1.2.2. We look up the current number of moves we have calculated for this corner state. Then, for each of the 18 moves from this state, we calculate the 18 different corner states that it can generate. For each of these new states we find the new corner encoding and check to see if the moveCount we calculate is less than our current ‘best guess’. If it is, then we need to add it to the work queue so that we can further expand this state as it may change other ‘best guesses’ we have for the shortest number of moves to solve a state. When the work queue is empty, it means we’ve found no states that can be reached in a shorter number of moves, which gives us the shortest number of moves for any corner state.

**4.1.2.6 The Search** Now that we have the pattern database generated, the the searching is fairly simple. Our search follows the typical IDA\* algorithm described in section 2.3.3 of the background. Firstly, we load our pattern databases into memory. This can take a few seconds but we keep the pattern database in memory for all susequent solves so we only have to do this once. We take the cube state and maximum depth to search as arguments. The maximum depth is 20 by default since this is God’s number: `public static String idaStarKorfs(int maxDepth, Cube cube)`. We take the cube state and use our perfect hashing scheme to number it. We can now lookup this number as an index into our pattern database to obtain the estimated moves to being solved. This is our first bound because we know that the number of moves required to solve the cube will be at least as much as this estimate.

Some implementations of Korf’s algorithm increment the bound when we fail to find a solution for that bound, but we can do better than this. Since the heuristic for Korf’s algorithm is admissible, we know that the real number of moves needed to the goal can not be better than the number of moves given by our heuristic. Assuming we’ve searched every node at a specific bound, we know that the solution length will be at least as long as the minimum estimate to the goal for every node. We can recursively track the minimum bound in our tree whilst we traverse it so we will not waste time attempting to sort through lots of heuristic values.

Figure 4.5: Main IDA\* loop

```
1 public static String idaStarKorfsStock(int maxDepth, Cube cube) {
2     ByteDeque solution = new ByteArrayDeque();
3     int result = 0;
4     if(Cube.isSolved(cube)) {
5         return "";
6     }
7
8     int bound = getH(cube);
9     while(bound < maxDepth) {
10         result = searchStock(cube, 0, bound, solution);
11         if(result == FOUND) {
12             return giveSolution(solution);
13         } else if(result == NOT_FOUND) {
14             return null;
15         } else{
16             bound = result;
17         }
18     }
19
20     return null;
21 }
```

We then proceed to try all 18 moves on our current state: R, R2, R3, U, U2, U3, etc. This will give 18 new states. For each of these 18 states, we must recurse. We first check if any of these states are solved. If one is solved, we return FOUND. For each of these states, we must recursively lookup the estimated number of moves to goal and then check if the current number of moves we've done added to the estimated number of moves to goal exceeds our current bound. If it does, we return the bound immediately for this branch which stops any further searching of this branch. Otherwise, we continue to recurse. Notice how at the end of each search, we must undo our last move using `cube.move(Cube.INV_MOVES[move])`. This is because we don't want to spawn a new child state in memory everytime we move, so we reuse the original cube object and just undo our previous moves to get back to our original state.

Figure 4.6: Search function

```

1 static int searchStock(Cube cube, int g, int bound, ByteDeque solution) {
2     //Check if the cube is solved
3     if(Cube.isSolved(cube)){
4         return FOUND;
5     }
6
7     //Estimate the number of moves to solve
8     int h = getH(cube);
9     int f = g + h;
10
11    //Return if the estimate is too high
12    if(f > bound){
13        return f;
14    }
15
16    int min = Integer.MAX_VALUE;
17
18    //Try every move
19    int t = 0;
20    for(int move = 0; move < Cube.NUMMOVES; move++){
21        //Try the move and add it to our solution stack
22        cube.move(move);
23        solution.addLast((byte) move);
24        //Search the subtree from this new state
25        t = searchStock(cube, g + moveCost[move], bound, solution);
26        if(t == FOUND){
27            return FOUND;
28        }
29        if(t == NOT_FOUND){
30            return NOT_FOUND;
31        }
32        if(t < min){
33            min = t;
34        }
35        //Undo moves and move onto next move
36        cube.move(Cube.INV_MOVES[move]);
37        solution.removeLast();
38    }
39    return min;
40 }
```

### 4.1.3 Improvements

Implementing Korf's algorithm, we can only make so many design decisions about the base Korf's algorithm until we can get no faster. Eventually, we need to make tweaks to the algorithm. The main drawback of Korf's algorithm is the time it takes to find an optimal solution. This is due to its average branching factor. Although a reduction from 18 to 13 is a significant reduction, a branching factor of 13 is still relatively large. In this section, we detail attempts at speeding up Korf's algorithm.

#### 4.1.3.1 Randomisation

Korf's algorithm uses a systematic depth first search until the branch it is searching reaches the specified bound, but why should we search systematically when the cube isn't scrambled systematically? Assume R is always the first branch, further assume we have a cube state which can be solved in 3 moves. What are the chances that 2 of 3 of these moves are an R move? Additionally, by searching systematically, solutions that start with moves involving the last branch will always take significantly longer to find than earlier branches.

Instead of systematically searching, we choose to explore branches randomly to eliminate this bias and give each branch an equal chance of being explored earlier. Additionally, this will speed up searches on average since we reduce the chance of searching unlikely branches that contain cube states reached by performing

many of the same moves.

#### 4.1.3.2 Branching factor reduction

Another way to speed up Korf's algorithm is to attempt to reduce the branching factor even further. We can do this by completely eliminating redundant move sequences. For example, if we've just searched a state that has just performed an R move, we should not attempt to search branches R, R2 or R3 from the current state. This is because performing another R move would be the same as performing an R2 move and performing an R2 move would be the same as performing an R3 move which we would eventually search anyway. Finally, the R3 move would just cancel out the previous R move.

We can extend this idea further to opposite sides. Imagine we perform the moves  $R \star L \star R3$ . Performing moves  $R \star R3 \star L$  would give exactly the same cube state, which can be further simplified to L. This is because opposite faces affect two disjoint sets of edges and corners so they can be performed in any order. Searching these sequences are clearly redundant.

To implement this, we iterate backwards over our solution stack. We look at the latest move and then rule out the relevant moves. We then look at the second to last move and check to see if that move involves the opposite face to the last move. If it does, we rule out relevant moves for the opposite face. We only need to look back 2 moves since the search is recursive. We can prove that any length of redundant sequence will be eliminated. Using a simple example:  $R \star L \star R3$  would never be a branch that we search because R3 would be ruled out once we see  $R \star L$  has already been performed.

Similarly, any redundant length sequence can be caught. E.g.  $R \star L \star R \star L \star L \star R$  would never be a branch we explore because  $R \star L \star R \star L \star L$  would also never have been generated in the first place because  $R \star L \star R \star L$  would have never been generated because  $R \star L \star R$  would have never been generated because R would have been ruled out when we see that  $R \star L$  has already been performed. A similar argument can be made for any length of redundant sequence. Notice how the redundant sequence is equivalent to  $R3 \star L3$  which will eventually be searched so we are aren't missing any branches that would potentially lead to a solution.

#### 4.1.3.3 Making use of memory

Since Korf's algorithm uses very little memory (as much as the depth of the search tree), we can put this memory to use elsewhere.

**4.1.3.3.1 Cache** Often performing the same sequence of moves over and over will give you the same cube state you've started with. For example:  $(R2 \star U2)^6$ . It is also possible that 2 different sequences of moves lead to the same state. It is obvious that we cannot store every state we've ever seen; the search space is far too large. This is why we have chosen to implement a fixed size cache that stores recently seen cube states. When the cache is full, we remove half of the oldest entries from the cache. This is because we do not want computation time to be dominated by removing and replacing cache entries each time we use the cache. The cache stores two things: cube state and number of moves used to get to that cube state. We add a cube state to the cache if it does not already exist there or if the cube state is reachable via a smaller number of moves. When we encounter a cube state that is already in the cache and takes more moves to get to that state, we choose not to expand this cube state any further because we know we've already explored a smaller sequence of moves to get to that state. Although this will not completely eliminate duplicates, it should have some impact on the duplicate state size reduction. We've found that a cache size of around 1,000,000 nodes works well with 6GB of RAM. We implement this using a Java HashMap to minimize lookup times for each state.

**4.1.3.3.2 Fringe Search** The major drawback with the IDA\* algorithms low memory consumption is that it is memoryless, i.e if it finds no solution for a specific bound, it will need to repeat all of its previous work in order to search to the next bound. This is because we pruned branches based on the the estimated moves to goal using the previous bound but the same branches may pass the heuristic tests for the new bound which means they need to be explored. The fringe search improves on this by storing 2 fringe lists[26]:

- **Current** - Stores cube states that we currently need to expand for the current bound.
- **Future** - Stores cube states that we need to expand on the next bound.

In a fringe search, we start the same as IDA\*. The only difference is that when we hit a cube state whose estimation to the goal exceeds the current bound, in addition to pruning the branch, we add that cube state to the ‘Future’ list. Once we finish exploring the bound, if we still have not found a solution, we move to the next bound. But instead of starting our search again, we move the ‘Future’ list that we collected last round into our ‘Current’ list. We iteratively expand nodes in the ‘Current’ list which simultaneously populates the ‘Future’ list for the next bound. This reduces search time since we won’t have to repeat the search from the top but instead we carry on where we left off in the previous bound. In essence, we are storing the ‘fringe’ of our search so we can pick up where we left off from the previous bound without having to repeat all of our previous work.

In our implementation, we only keep track of solutions in our ‘Current’ and ‘Future’ lists. This is because as long as we have a sequence of moves and the initial scrambled cube state, we can recover any state we were previously on by simply performing the sequence of moves we have in our potential solution. Fringe Search has been proven to be faster than A\* and IDA\* in path finding game maps [26]. We wanted to experiment with it on an application with a much larger branching factor because the size of our lists will explode much faster. We perform a few tests in the Evaluation chapter to see how far we can take Fringe Searching.

#### 4.1.3.4 Parallelism

**4.1.3.4.1 IDA\*** We can improve the speed of the IDA\* search by parallelising the search. Assuming we have  $n$  threads, we can split the tree into  $n$  subtrees and search each subtree in parallel. We find that using one thread per core yields the best results. Figure 4.7 shows how we can split the tree.

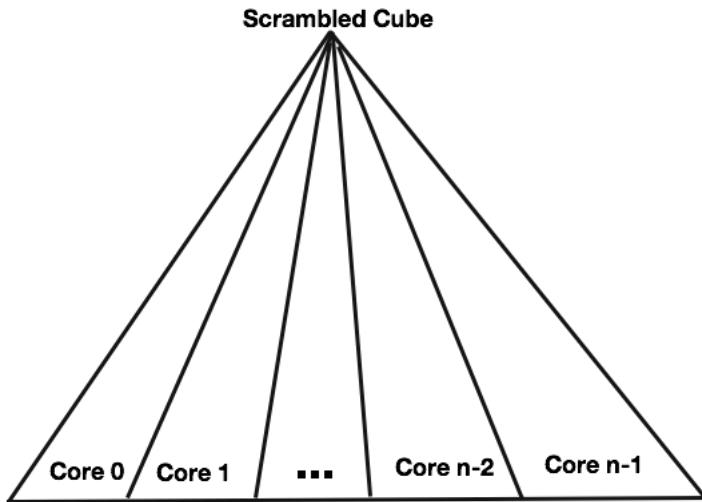


Figure 4.7: How we split our search space

**Synchronisation** We synchronise all  $n$  searches so that all searches search with the same bound. Even if one core finishes early, we must wait for the other cores to finish their searches first. This is because even if we found a solution in some higher bound, we would need to wait for the other cores to finish their searches for the lower bounds in order to prove that the solution found is optimal. However, if we do find a solution

at some bound and all cores are searching at that bound, we can terminate the search early!

To do this, we use an array containing n flags. Each thread will have its own flag. A main thread is responsible for keeping track of whether or not a solution has been found. It will set off each thread to search their own respective subtrees and then sleep. If a thread finds a solution, it will set its flag to FOUND before waking up the main thread. The main thread will then wake up and see that a solution has been found and terminate all other n-1 threads. On the other hand, if none of the n threads finds a solution, they will all set their flags to MOVE\_ON and wake up the main thread. The main thread will wake up and see that all threads have asked it to move on. The main thread will only move on if ALL of the n threads have finished their search.

**4.1.3.4.2 Fringe** We also experimented with multithreaded fringe search. In our multithreaded fringe search, we expanded a node in the fringe just like how we parallelised our IDA\* algorithm. That is, we split the search tree n times using the fringe node as our root. This means we have to synchronise access to our ‘later’ fringe list since there will be lots of concurrent writes to this list from the n threads.

#### 4.1.4 HPPC Java Library

The problem with Java collections is that they only work for non-primitive types/classes. This means when we need collections of primitive type `int`, we will need to use the `Integer` Wrapper instead. A primitive `int` uses 4 bytes but an `Integer` uses 16 bytes. This is a disaster for memory conscious applications! Fortunately, the HPPC (High Performance Primitive Collections) Library provides custom Java collections for primitive types. IBM’s experiments with HPPC compare a ‘Textbook’ Java A\* algorithm implementation vs a HPPC implementation. Staggeringly, it managed to reduce a 25GB memory consumption to just 7GB![10]

We make use of HPPC’s `ByteArrayDeque` collection to keep track of our solutions. This collection becomes invaluable when we implement fringe search in section 4.1.3.3.2 which must keep track of many solutions in a list. The `Byte` wrapper uses at least 8 bytes of memory so the classic Java alternative would have cost a lot of memory.

## 4.2 Kociemba’s algorithm

Kociemba’s algorithm is the most popular algorithm for a couple of reasons: it is reasonably close to optimal solutions and its speed of finding solution. We have used Kociemba’s Java and C libraries for Kociemba’s algorithm.[14]. Here we describe the main differences in Kociemba’s implementation from our implementation of Korf’s algorithm.

### 4.2.1 Coordinate Labelling

As mentioned in section 2.3.4, Kociemba’s algorithm merges together groups from Thistlethwaite’s algorithm which causes the groups to be so large that we can’t just generate move tables to get from one group to another. Instead, we must perform a search to move to the other group. Most implementations use IDA\* as the search method. This means we need to generate pattern databases. The pattern databases generated are slightly differently to Korf’s algorithm since our goal is not to solve the cube in one step, but two. We also need to modify our minimal perfect hash.

#### 4.2.1.1 Phase 1

Remember that the first step moves us from  $G_0 = \langle R, U, F, L, D, B \rangle$  to  $G_1 = \langle U, D, R2, L2, F2, B2 \rangle$ . This step makes all corners and edges ‘good’ and puts edges that belong on the middle slice between faces U and D (UD-slice for short) within that middle slice. We can use our numbering system described in Korf’s algorithm (but slightly differently) in order to number these states, just like how we did in Korf’s to save space.

We can represent any corner orientation using an integer between 0 and 2186 (see section 4.1.2.2.1). Edge

orientations can be numbered between 0 and 2047 because an edge orientation is either 1 or 0 so we can describe any total edge orientation state using a combination of 11 1's or 0's. We only need 11 states because even though we have 12 edges, the edge flip lemma described in section 2.2.1.2 halves the number of reachable edge orientations to  $2^{11}$ . Finally, the number of edges on the UD-slice is 4. That means that these edges can appear anywhere within the 12 edge positions and gives us a total number of UD-slice numberings of  $12C_4 = 495$  so we can number each of these UD-slice states with a number from 0 to 494. We can then combine this triple to give us a unique encoding for any state in phase 1:

$$\text{Phase1Encoding} = (\text{CornerOrientationNumber} * 2048 + \text{EdgeOrientationNumber}) * 495 + \text{UDsliceNumber} \quad (4.4)$$

This means our Phase 1 pattern database will be of size  $2048 * 2187 * 495 = 2,217,093,120$ .

#### 4.2.1.2 Phase 2

The phase 2 step moves us from  $G1 = \langle U, D, R2, L2, F2, B2 \rangle$  to  $G2 = \{C\}$ , i.e solve the cube. In this step we want to permute all of our remaining 8 edges not in the UD-slice and corners and we want to permute our UD-Slice edges in their correct positions. We know that all edges and corners are orientated to make them ‘good’. Again, we can number our edge permutations using the same method as our implementation in Korf’s algorithm but this time we only have 8 edges and 8 positions for those edges. Since we can only use moves U,D,R2,L2,F2 and B2, the 4 positions in the UD-slice are not reachable but these 8 remaining edges. This gives us  $8! = 40320$  states which can be numbered from 0 to 40319 using our factorial numbering scheme. Permuting the 8 corners also has  $8! = 40320$  states since there are 8 corners and 8 positions that those corners can reach using the moves in G1.

Moving the UD-Slice edges into their correct positions only has  $4! = 24$  states since there are 4 edges on this UD-slice and the no edges can leave or enter the UD-slice. We can use our factorial numbering scheme to number these states between 0 and 23. We can now label any state within phase 2 uniquely:

$$\text{Phase2Encoding} = ((8\text{EdgePermNumber} * 40320) + \text{CornerPermNumber}) * 24 + \text{UDslicePermNumber} \quad (4.5)$$

This makes the Phase 2 pattern database a size of  $40320 * 40320 * 24 = 39,016,857,600$

#### 4.2.1.3 Two-Phase

Just like Korf’s algorithm, now that we have a way to uniquely number each important state of the cube, we can now generate our pattern databases using our work queue approach in section 4.1.2.2.5. Armed with 2 pattern databases for each phase, we can perform a search from  $G0 \rightarrow G1$  and then  $G1 \rightarrow G2$  using IDA\*.

## 4.3 Combining Kociemba’s and Korf’s

Now that we’ve implemented both Korf’s and Kociemba’s algorithm, let’s see how we can combine them. We know that Korf’s algorithm is guaranteed to give us an optimal solution but has an unpredictable time for termination. Given a scrambled cube, we won’t know how long Korf’s algorithm will take to find a solution. On the other hand, we know that Kociemba’s algorithm returns a non-optimal solution but within a reasonable time. (Under 1 second (section 5.2.1)). This means we can use Kociemba’s solution as an upperbound estimate so that we can perform a more informed Korf’s search for a potentially better solution with the remaining time. Within this time, there are three cases which can occur:

1. **A better solution is found** - In this case, we replace the solution found by Kociemba’s algorithm with the one found by Korf’s algorithm.
2. **Nothing is found** - In this case, we just use Kociemba’s algorithm’s solution.
3. **A solution of the same length is found** - In this case, we calculate a rough ‘time to solve’ estimate for both solutions described in section 4.3.1, and pick the faster solution.

### 4.3.1 Time to Solve Estimation

We can compare two sequences of moves by estimating the amount of time they would take to solve. Although two sequences may have the same length, the sequence that requires less cube rotations is preferred. Since we've chosen to use a four armed robot, there are two faces: U and D, that we cannot directly turn without cube rotations. We must rotate using either an X rotation or a Z rotation. If we perform an X rotation, any F or B moves would now require a cube rotation. Likewise, if we perform a Z rotation, any R or L moves would require a cube rotation. We track the orientation of the cube and use profiling estimates to estimate how long each move will take to perform and then sum the estimates.

**4.3.1.0.1 Profiling** We profile by simply measuring how long each move takes to execute in different orientations and then rounding to give a rough ratio. A quarter move that requires no rotations is of size 1. Everything else is then measured relative to this. We are essentially using quarter move rotations as a unit of measurement.

## 4.4 Searching for shortest number of robot moves

Although Korf's algorithm finds an 'optimal' solution in the sense of the number of face turns required, it does not find the fastest solution possible for our robot. We experimented with a variation of Korf's algorithm that would search for the fastest solution the robot could perform. This is a great idea in principle, but early benchmarks revealed that this would increase search times by too much so we decided to not continue further with this idea for this project. Although unsuccessful, we feel it is useful to show how we implemented the search algorithm and what caused the idea to fail should we want to continue with this idea in the future.

### 4.4.1 Dynamic Costing

In the normal Korf's implementation, we make the assumption that all moves carry the same weight, i.e. all moves are equivalent in cost. In reality this isn't true. On our 4 armed robot, U moves take much longer to perform than R moves. This is because in order to perform a U move, we must rotate the entire cube using a Z or an X rotation. By using the profiling technique in our 'Time to Solve' estimation above, we were able to determine that moves requiring cube rotations take around 3 times longer than those that do not. A more detailed breakdown of how we perform these rotations can be seen in section 4.6.2.

Imagine we perform an X rotation in order to perform a U move. All subsequent U moves can now be executed quickly. However, how much is a B or F move now going to cost? They cost around the same as U moves used to. This is because we now need to perform an X3 rotation in order to perform B or F moves. This shows that our costs are not static as we assume in Korf's algorithm, but they change depending on the orientation of the cube. That means we will need to track an extra piece of information: cube orientation.

There are three possible orientations that we need to consider, the standard orientation we have been working with so far, the orientation we obtain from performing an X rotation and the orientation we obtain from performing a Z rotation. Not only this, there is now more than one branch for each move. All moves on the U or D can be performed after a Z rotation or an X rotation which would lead to different costs for subsequent moves. This increases our branching factor from 18 to 24. The same reasoning can be applied to any other face for other orientations of the cube.

### 4.4.2 Reducing the branching factor

To combat this massive increase in an already huge branching factor, we made the assumption that we could only perform X or X3 rotations. This reduces the number of options available for a U turn back down to a single branch per move. Although this would no longer give the shortest number of moves the robot would need to make, we were sure that this would still give us fairly short move solutions.

### 4.4.3 Search speed

We made a few early benchmarks for small scramble lengths of 10, 12 and 14 just to test how long searching would take using this dynamic costing so that we could measure its viability for use in this project. The results were unexpected. Searching for solutions for scrambles of length 10 took 40 times longer to find over the normal Korf's algorithm. In addition, these solutions were no different from Korf's algorithm meaning at depth 10, all of our test scrambles gave an optimal solution that was also the shortest number of robot moves! At a depth of 12, the algorithm took a few hours to find a solution. Again, the solution was exactly the same as the normal Korf's algorithm.

#### 4.4.3.1 Why were solutions the same?

An explanation as to why this may be happening is because we tested our algorithm at small scramble depths. At scramble depths this shallow, there is almost always only going to be one solution of optimal length. Any suboptimal solutions will need far more moves to solve. Why is this? When the cube is in a solved state, each successive move that we perform will take it further and further away from the solved state. That is, until we hit God's number, 20[18]. We know that any moves we perform on a cube state that is 20 moves away from being solved cannot possibly increase the number of moves needed to solve the cube. In fact, if we make a move from a cube state that is 20 moves away from being solved, it is guaranteed that we either decrease the number of moves required to solve the resulting state by 1, or a different set of 20 moves are required to solve it.

Here is a simple proof. Assume we have some cube state  $S$  that requires 20 moves to solve optimally. We know that if we move any face, the number of moves to solve the resulting state must be at least 19. Why can it not be below 19? If we arrive at a state that needs less than 19 moves to solve and we 'undo' the move we just performed, our solution length would be at most 19, which contradicts the assumption about us needing 20 moves to solve the cube optimally. This shows that there could be multiple close to optimal solutions for long solution depths.

Let's assume that our move,  $M$ , on  $S$  gives a resultant state that requires 20 different moves to solve optimally. As well as the 20 moves, we have another solution of length 21:  $(S * M)^{-1}$ . If the difference between our optimal and suboptimal solution is just 1, we can see that it is easy for us to solve the cube faster than the optimal one if we have 1 less cube rotation in our suboptimal solution.

At lower scramble depths, the story is very different. Any move we perform can either increase or decrease the number of moves we will require to solve the resulting state, so the probability of there being a solution that is only 1 or 2 moves away from optimal and has less cube rotations than the optimal solution is lower than at longer solution depths.

#### 4.4.3.2 Why did it take so much longer?

As mentioned, moves that require cube rotations take around 3 times longer to perform. Our bounds for Korf's algorithm move in increments of 1 the majority of the time. Our bounds for this variation remain moving in increments of 1 but our costs are rising much faster. A solution that used to only cost 10 could now cost 14 if we penalise cube rotations. 14 is much deeper in the search tree than 10 so it will take us much longer to find this solution even though it will be the same sequence of moves.

#### 4.4.3.3 Why is it not viable?

The increase in search depth and low probability of occurrence means that we are spending more CPU time for something that isn't likely to happen. If we use the algorithm to search for depths around 20, the algorithm will be just as slow (taking a few hours). Having the normal Korf's algorithm terminate within a reasonable time is a big task, so to have this algorithm terminate within a reasonable time for any normal scramble depth is a whole new problem in itself.

## 4.5 Vision System

The vision system has a wide scope of potential. We could have taken a simple approach to extracting the colours from the cube whilst sacrificing robustness and reliability. However, we chose to put more time into making the vision more robust and reliable since this is the start of the cube solve. If we take a lot of shortcuts with the vision system, we reduce the reliability of the system as a whole since a cube solve cannot start if we cannot reliably determine the state of the cube.

The vision system has three stages:

1. **Cube recognition** - Recognise where the cube is in the frame and identify where the stickers lie.
2. **Colour extraction** - Accurately determine the colour of the area that we have identified as being a sticker.
3. **Construct the cube state** - Construct the cubestate and check if the cube state is a valid one according to lemmas from section 2.2.1.

### 4.5.1 Cube Recognition

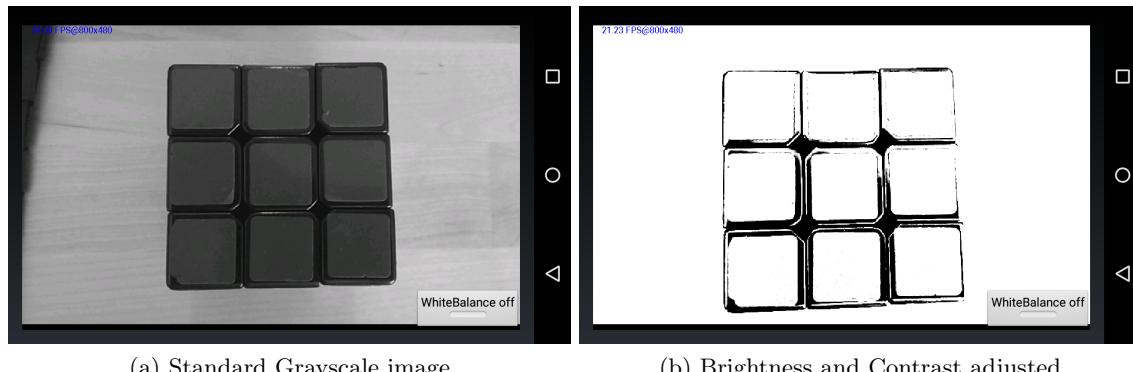
The first step to our vision system is attempting to detect the cube. But what makes a Rubik's cube a Rubik's cube? The sequence of transformations to an image to detect the cube are as follows:

1. Convert image to grayscale
2. Adjust brightness and contrast
3. Gaussian blur
4. Laplacian operator
5. Dilation
6. Adaptive thresholding
7. Find contours

#### 4.5.1.1 Contrast and Brightness Adjustment

We take this step to make edges more obvious for when we wish to find important edges of the image. We up the brightness to account for the blue stickers which are quite dark and often mistaken for black. By increasing brightness and contrast, we increase the difference between black and blue and reduce the chance of misinterpretation.

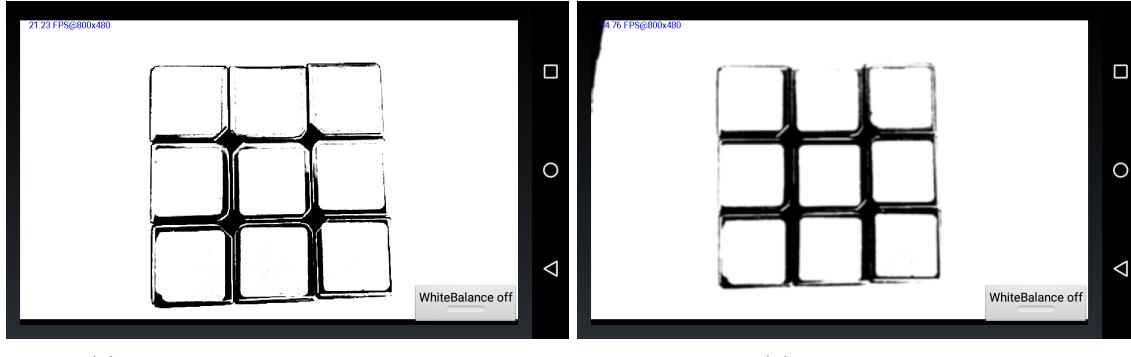
Figure 4.8: Contrast and Brightness Adjustment



#### 4.5.1.2 Gaussian Blur

By using Gaussian Blur first we can remove noise. We've found Gaussian Blur with these parameters quite effective at removing noise for a Rubik's Cube:

```
Imgproc.GaussianBlur(mat2, mat2, new Size(7,7), 0);
```



(a) Brightness and Contrast adjusted

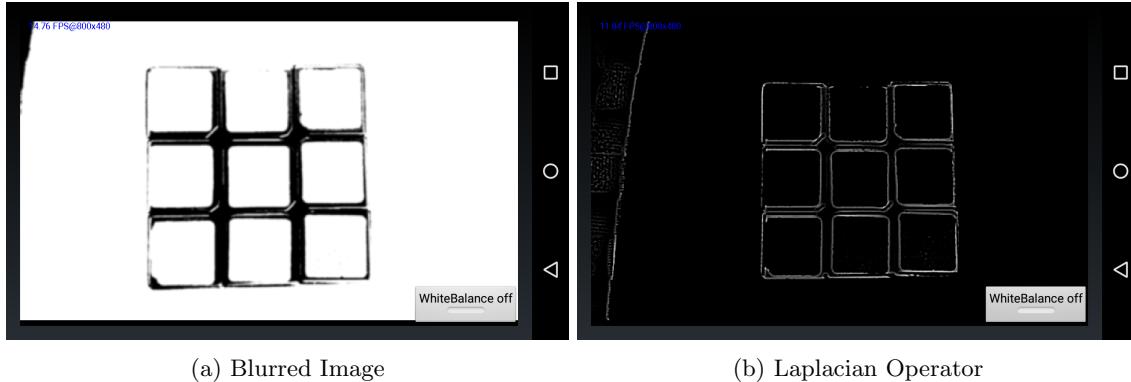
(b) Blurred Image

We can see that Gaussian Blur removes a lot of noise caused by light reflections. Since the cube is plastic and the stickers are glossy, this step is crucial to get clean edges for the next step.

#### 4.5.1.3 Laplacian Operator

As described in 2.4.3.1, Laplacian Operator is used to detect edges of an object by looking for sudden changes in pixel intensity. We've found that the Laplacian Operator with these parameters works well for finding the edges of the Rubik's Cube:

```
Imgproc.Laplacian(mat2, mat2, CvType.CV_8U, 3, 1, 0, Imgproc.BORDER_DEFAULT);
```



(a) Blurred Image

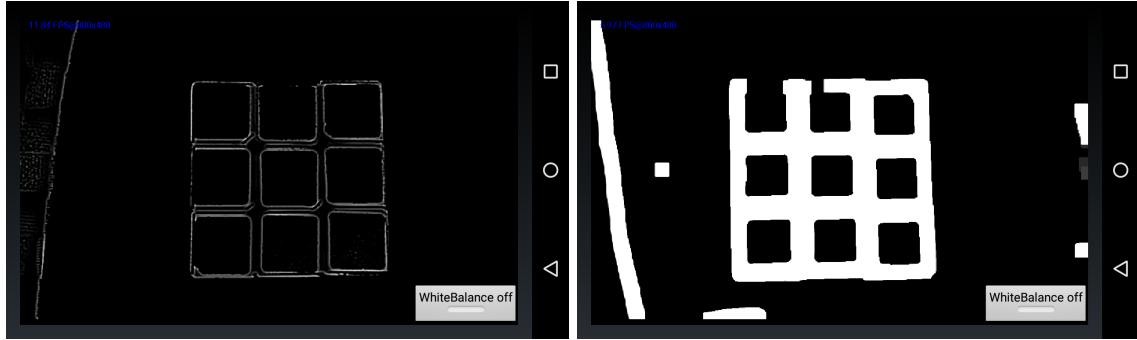
(b) Laplacian Operator

We can see that Laplacian Operator gives us a vague outline of the cube and each of the stickers but also gives us a few spurious outlines from noise that was not removed during the Gaussian Blur phase as well as outlining other objects in the frame.

#### 4.5.1.4 Dilation

Since the Laplacian Operator gives edges that are rather thin, we use a dilation transformation to make these lines extremely obvious. That way, we can extract square regions of the cube with confidence:

```
Mat size20 = Imgproc.getStructuringElement(Imgproc.MORPH_RECT, new Size(20, 20));
Imgproc.dilate(mat, mat, size20);
```



(a) Laplacian Operator

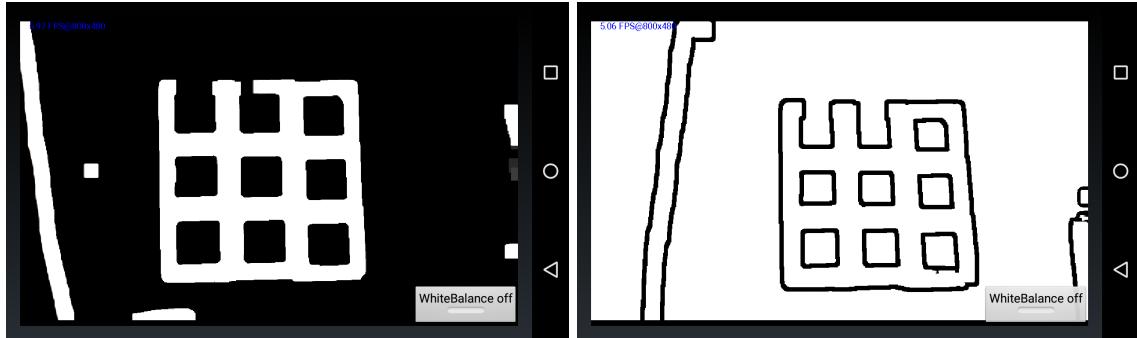
(b) Dilation

We can see that dilation merges together some noise around the cube edges to eliminate them. We also see that other objects edges that are not the cube also dilate! This is not ideal.

#### 4.5.1.5 Adaptive Threshold

Adaptive Thresholding allows us to outline the regions where the stickers lie. Usually, Adaptive Thresholding converts a grayscale image to a binary image, but if we use a small neighbourhood value, it can act as a very accurate edge detection technique. We have found that using Adaptive Thresholding like this works very well:

```
adaptiveThreshold(m, m, 255, Imgproc.ADAPTIVE_THRESH_MEAN_C, Imgproc.THRESH_BINARY_INV, 15, 4);
```



(a) Dilation

(b) Adaptive Threshold

Notice how it outlines other objects in the frame too! We need a way to only recognise the cube in the image.

#### 4.5.1.6 Find Contours

Contours are boundaries on an image and need to be closed curves. Now that we have an image where we can reliably extract closed curves, we use:

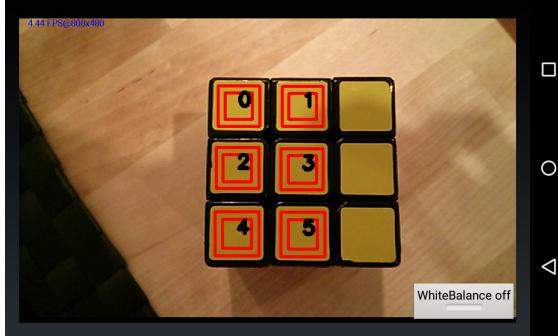
```
findContours(m, cont, hier, Imgproc.RETR_LIST, Imgproc.CHAIN_APPROX_SIMPLE);
```

This gives us a list of closed curves in the image which eliminates most of the spurious edges. We still have a problem. What about the closed curves that aren't part of the cube? We only want contours that are shaped like a square! We can detect squares by drawing a bounded rectangle around each of the contours and check for a few properties:

- Height and Width of bounding box are approximately the same.
- Difference between inner area of contour and bounding rectangle area is below a threshold. We've found that below 10% gives a good estimate.

We still have another problem. What if there are other squares in the image? We eliminate other squares in the image by looking for only squares that are the same size as the Rubik's cube stickers. But how do we know how big the stickers are? We could assume some threshold but this would restrict the distances that we would be allowed to have the camera at too much. Instead, we assume that the majority of squares will probably come from the Rubik's cube itself and look for the median square's area. If the number of squares is dominated by the Rubik's cube, the median area value should be one of the Rubik's cube sticker. We only proceed to the colour extraction stage if we have all 9 squares detected in a grid.

Figure 4.13: Finding Stickers

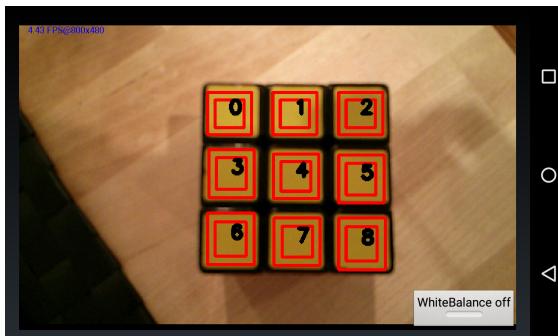


(a) Identifying Stickers on the cube

#### 4.5.1.7 Square Recovery

We can see that the sticker detection in section 4.5.1.6 works well but occasionally misses some stickers. This often comes from light reflecting on the sides of the Rubik's cube that makes the black plastic look white. We work around this by trying to recover the remaining position of the stickers. We can predict where any sticker of the cube lies by simply having any number of stickers that can give us the minimum and maximum ( $x, y$ ) coordinates for the cube. For example, 2 opposite diagonal corners.

Figure 4.14: Finding Stickers



(a) Recovering the other squares

**4.5.1.7.1 Prediction** We estimate the distance between any two stickers by using the height or width of the bounding box. We've found that 20% of the bounding boxes width added to the bounding boxes original width is a good estimate. From here, since the stickers are arranged in a grid, we can recover any other sticker position by simply adding or taking away our distance between stickers' measurements on the x or y axis and since we have maximum and minimum ( $x, y$ ) coordinates, we know exactly where each sticker lies relative to another.

## 4.5.2 Recognising Colour

Now that we know the positions of each of the stickers on the face, we can begin to extract its colour. The major difficulty with this stage is the thresholds needed to determine colour.

### 4.5.2.1 RGB vs HSV

We've seen two representations of colour in section 2.4.1. We've found that representing colour as HSV is a more robust method for detecting colour. When using RGB, the thresholds change too drastically depending on the brightness of the image and the colour of the light source.

### 4.5.2.2 Extracting Colour

We first construct an inner box that lies within our bounding box. This is to lower the chance of picking up the black plastic (from the cube) around the sticker. We've found that around 75% of the original bounding boxes' size gives a region of the sticker that is large enough so that we can reliably determine its colour but not so large that we would see any black plastic. To recognise colours in this region, we first create a histogram of our hue channel for each sticker. Our histogram ranges from 0 to 180 and our buckets are of size 1. This will give us 9 histograms with a count for the number of times a hue in a pixel lies within a specific range. For any given sticker's histogram, we search for the bucket that has the biggest value. This will be the hue that appears most frequently in the sticker. We then find the mean saturation and value for the other 2 channels. We found that finding the most frequent hue worked a lot better than the mean hue because it is not as easily affected by small regions of glare or noise. When we used the mean for hue, the mean hue colour would be a colour that didn't exist in the sticker region if there happened to be noise or glare in the image.

**4.5.2.2.1 HSV Colour Wheel** The HSV Colour wheel (Figure 2.16) gives us the degree thresholds for colour when our colour source is white. We used these thresholds to determine the colour of each sticker and assume we have white light. You may notice that the colour white itself is missing from the wheel. Remember from section 2.4.1.1 that colour is represented as 3 components in the HSV model. The colour wheel only gives thresholds for a hue with max saturation. White is just any colour with a high value and low saturation.

Figure 4.15: HSV colour thresholds

```
1  private FaceColour scalarToColour(Scalar hsvSc) {
2      int hMax = 179;
3      int sMax = 255;
4      int vMax = 255;
5
6      double[] hsv = hsvSc.val;
7
8      if (hsv[1] < 0.3 * sMax && hsv[2] > 0.3 * vMax) {
9          return FaceColour.W;
10     }else {
11         double deg = hsv[0];
12         if (deg >= 0 && deg < 5) return FaceColour.R;
13         else if (deg >= 5 && deg < 20) return FaceColour.O;
14         else if (deg >= 20 && deg < 45) return FaceColour.Y;
15         else if (deg >= 45 && deg < 90) return FaceColour.G;
16         else if (deg >= 90 && deg < 120) return FaceColour.B;
17         else if (deg >= 120 && deg < 180) return FaceColour.R;
18         else return null;
19     }
20 }
```

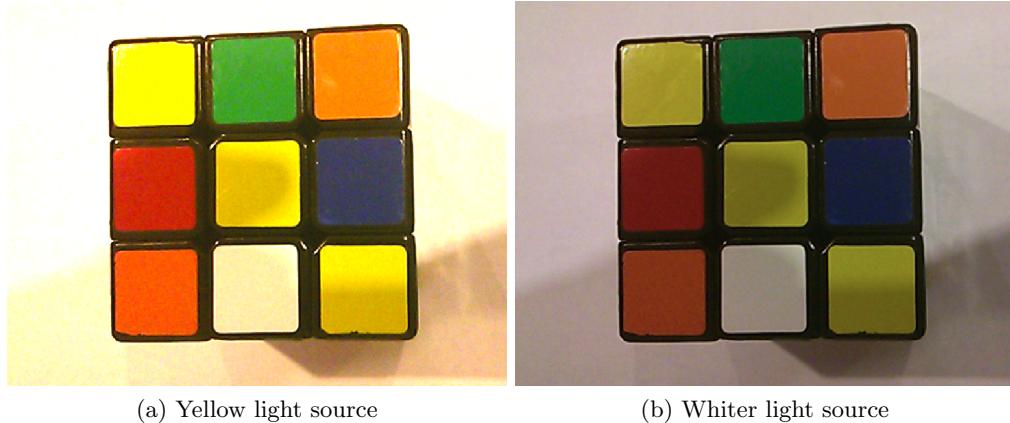
We've found that anything below 30% of the maximum S value for OpenCV and anything above 30% of the max V value identifies white stickers the majority of the time in a moderately lit room. The thresholds

for other colours will vary depending on the specific stickers on the cube.

#### 4.5.2.3 Grey World Assumption

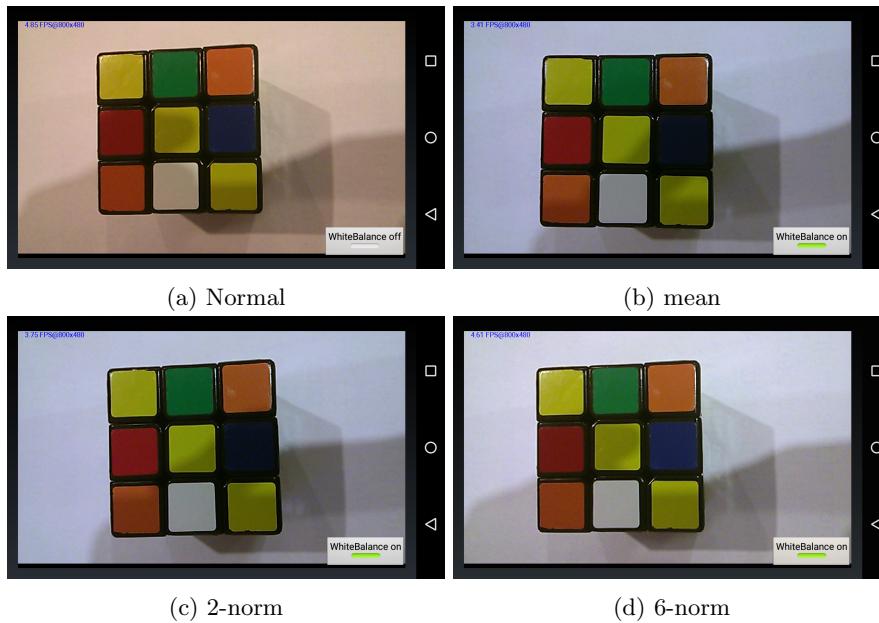
The HSV Colour wheel threshold is great when we have a natural white light source illuminating the cube. In most cases, this will not be the case, especially indoors where a lot of light is yellow. As mentioned in section 2.4.4.1, Grey World Assumption is a colour balancing algorithm that allow us to balance the colours back to as if they were illuminated via a white light source. We found that we were able to make this assumption because the majority of the time, each face of the cube will have a good distribution of colour.

Figure 4.16: Yellow vs white light



As we can see in Figure 4.16a, a yellow light source can have a profound affect on the hue and shifts all colours towards yellow. This is particularly prominent on the white sticker which now looks more like a yellow sticker.

**4.5.2.3.1 P-Norms** We experimented with a few different P-Norms as described in section 2.4.4.1.3 and found varying results between just a standard mean and a high P-Norm. In all cases we found that using the max average channel value mentioned in section 2.4.4.1.3 would yield a more well balanced image.



Using a normal mean, we can see that the Grey World Assumption algorithm tends to overcompensate the yellow light and now there is a blue illumination in the image. However, using a high P-Norm of around 6, we can see the Grey World Assumption algorithm balances the colour well enough that it looks like white light illumination!

## 4.6 Robot

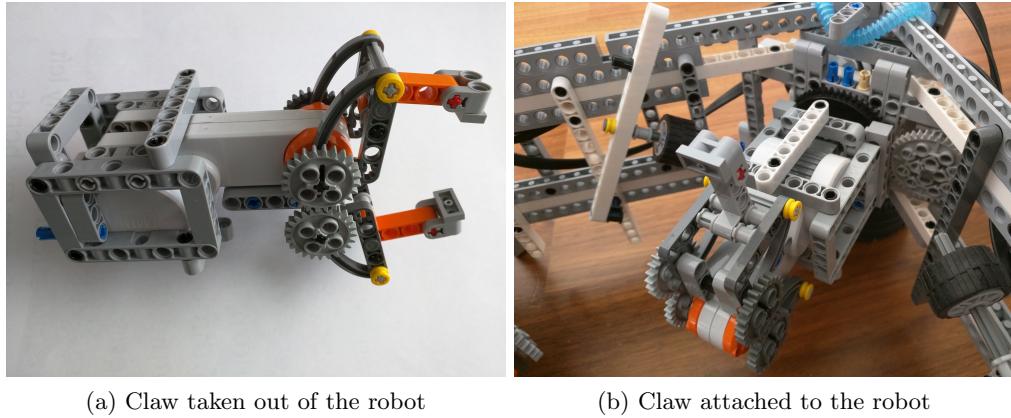
This section describes the final component of the system: The Robot. The Robot has a 4 arms in a master-slave configuration.

### 4.6.1 Hardware Design

#### 4.6.1.1 The Arm

Each arm is powered by two motors giving 2 degrees of freedom. One motor controls the the arm's clamping mechanism and the other allows the whole arm to rotate. Our setup consists of 4 of these claws meaning we need 8 motors in total.

Figure 4.18: Two different views of the claw



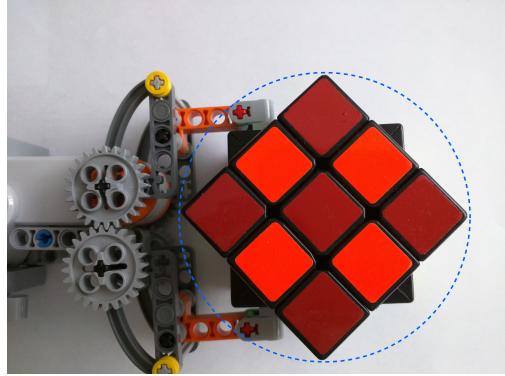
(a) Claw taken out of the robot

(b) Claw attached to the robot

One gear is connected directly to the motor whilst the other lies beneath it. Rotating the main gear on the motor in one direction will rotate the gear beneath it in the opposite direction which allows us to open and close the claw. Although the arm configuration looks fairly simple, there are a few important aspects to the arm that need to be considered:

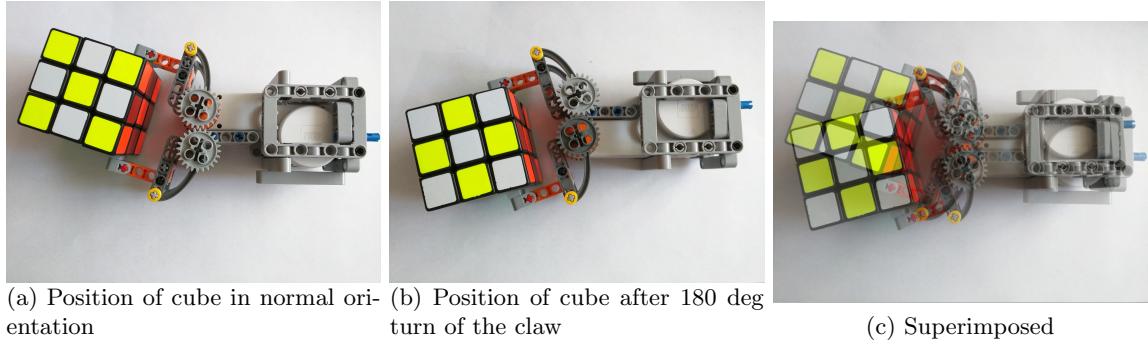
- **Clearance** - Adjacent perpendicular cube rotations must be able to be rotated freely without hitting this arm. Figure 4.19 shows this. We can see the outlined turning circle of the cube. The arm (excluding the claw) must be able to clear this turning circle.
- **Symmetric** - The clamp must close symmetrically so that the cube lies perfectly in the centre of the claw. This is so the centre of rotation remains consistently around the same spot. Small variations in the centre of rotation can lead to major shifts in the cubes position which would throw off all other arms. We achieve this by using two gears of the exact same size. This means they will both move at the same angular velocity.
- **Rigid** - The arm needs to be rigid enough to not warp under strain. We have reinforced the mounting points of the arms with square frames to prevent the arm from sagging when holding the cube. We also used 4 gears instead of just 2 to minimise gear slippage under high tension. Gear slippage can throw off the symmetry described above.

Figure 4.19: Turn circle of the cube



In figure 4.20c we can see the impact of having a non-symmetric claw gripping mechanism. The figure shows where the cube lies before and after a 180 degree rotation for a non-symmetric claw grip - a massive variation.

Figure 4.20: Two different views of the claw



#### 4.6.1.1.1 Performing a face rotation

To perform a face rotation, an arm does the following:

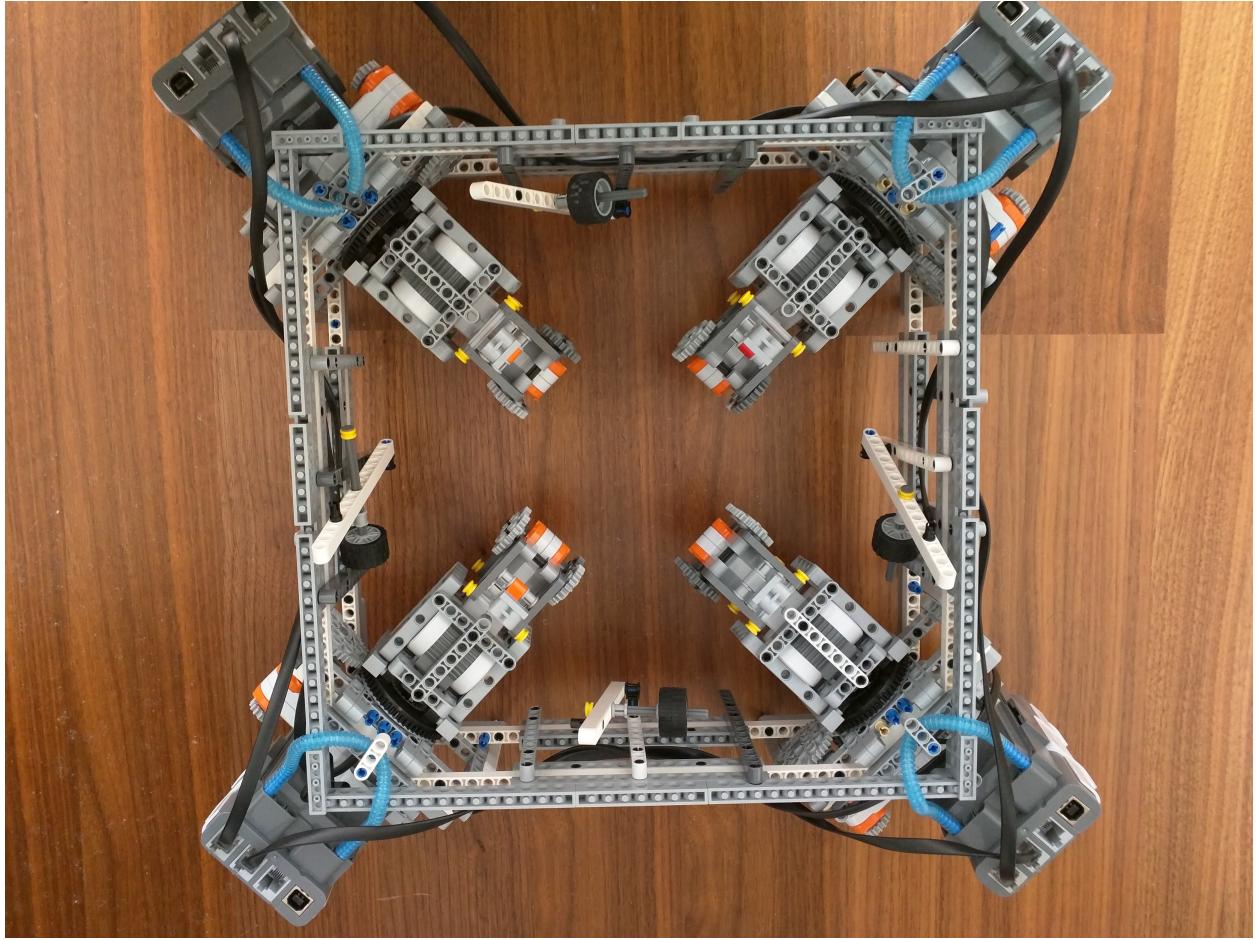
1. Clamp down on the face to rotate using the clamping motor
2. Rotate the claw that is now clamping the cube
3. Unclamp the face
4. Rotate the claw in the opposite direction back to its original position

**Why even unclamp to rotate back?** If we do not rotate the cube back, the wire that connects to the NXT motor will keep twisting until it becomes tangled. We always twist back in the opposite direction to reset the wire back to a neutral position.

#### 4.6.1.2 Robot Overview

Our robot uses 4 NXT Intelligent Bricks. Although 3 Bricks can control up to 9 motors and would easily accommodate 8 motors, the motors would divide awkwardly amongst the Bricks meaning we would need to write a lot of special cases for specific Bricks. We wanted to be able to run the same program on every Brick without having to write a lot of special cases so having 2 motors on each of the 4 Bricks was a much better option. Figure 4.21 shows a birdseye view of the robot.

Figure 4.21: Overview: Birdseye view



**4.6.1.2.1 Configuration Choices** Now that we have 4 Bricks and 8 motors, we need to decide which motors should be connected to which Bricks. Although this may seem like a trivial task, the choice makes a huge difference in how we implement the rest of the robot. We have 4 motors that control clamping (one for each arm) and 4 motors that control face rotation (one for each arm).

**By Arm** Each arm uses 2 motors. A natural way to divide by the 8 motors is to give each arm an independent NXT Brick. This gives us fine grained control over each arm and is easy to control using a master-slave configuration. We could only need to send a single bluetooth command to the Brick in charge of an arm to rotate a face since this single Brick can perform both clamping and rotation for that face. However, this configuration does bring a major complications of synchronisation with it. Rotating the entire cube needs two opposite arms to rotate in the same directions at the same speed. This would require synchronisation between the two Bricks that control both arms. Remember that we are restricted to a master-slave conuguration. The delay between the master and slave communication is never consistent and so synchronising between two slaves is complex.

We did experiment with this configuration and tried the Berkeley clock synchronisation algorithm[23]. We would attempt to sync the clock of the slaves with the clock of the master (our Smartphone). When the master sent commands to each of the slaves, it would also send a time for this command to be executed. Determining the time to execute was the hardest problem. We needed to assume that bluetooth had some maximum latency such that after this time, all messages will have been received. In reality, this assumption

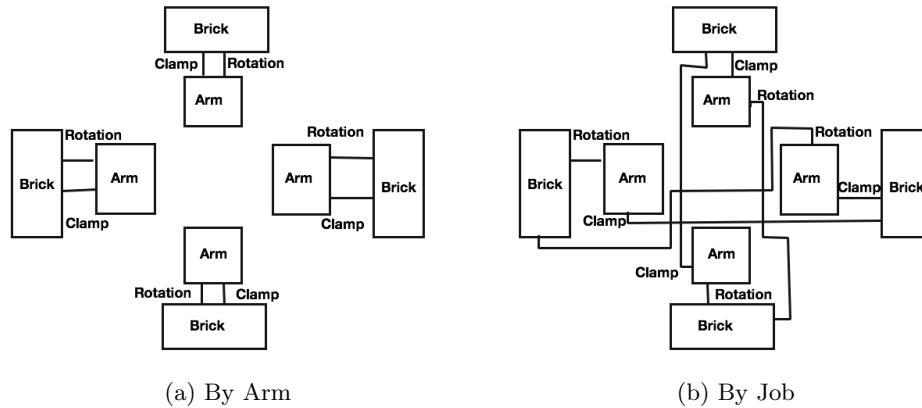
just isn't true so this configuration would fall over in cases where a slave would receive a message later than our specified maximum latency. We could keep increasing this maximum latency time but this would slow down the solve.

**By Job** A slightly less natural way of segmenting the motors is to separate by job. By 'job' we mean whether a Brick should be responsible for clamping motors or rotating faces. Each Brick can only control 2 clamps or 2 rotation motors. If we think about how we want our arms to operate, the only time where we would need 2 arms to rotate or clamp at the same time is when we perform a whole cube rotation. In this case, the two arms are opposite to each other. Therefore, it is logical that the 2 clamp or rotation motors that we connect to each Brick are opposite to each other. If a slave is responsible for clamping or rotating two opposite arms, we do not have to worry about synchronisation over bluetooth!

This does however, slightly complicate turning a face. We would need to send at least 4 messages (one for each instruction specified in section 4.6.1.1.1) to 2 Bricks. The Brick that is responsible for clamping the face we wish to turn and the Brick that is responsible for the rotation of the arm we wish to rotate.

Although segmentation by Job makes the face turning protocol more complex, it saves us from having to run a synchronisation protocol all the time. Most synchronisation protocols attempt to delay performing a job from the master until we are certain that all slaves have received their jobs so that we can start at the same time. This would slow down solve times since we would always need to synchronise and check other slaves before we can perform any moves. Therefore a small gain in complexity in this case is a good trade off for the gain in speed and is actually more simple when compared to a complex synchronisation protocol.

Figure 4.22: Configuration by Arm vs Job



## 4.6.2 Movements

With this design, we can perform any R, L, F or B moves with ease, since the arms clamp down on these faces by default. Let us label each claw ClawR, ClawL, ClawF and ClawB with respect to the default face it clamps down on. Let us also label the Bricks as follows: RLClamp, RLRotate, FBClamp, FBRotate, where RL or RB tells us that this Brick is responsible for those 2 faces and Clamp or Rotate tells us its job. Assuming the robot is already holding the cube (all clamps are closed), to perform a face rotation we do the following using R as an example:

1. Tell Brick RLRotate to rotate ClawR clockwise 90 degrees
2. Tell Brick RLClamp to release the clamp of ClawR
3. Tell Brick RLRotate to rotate ClawR anticlockwise 90 degrees
4. Tell Brick RLClamp to close the clamp of ClawR

This is pretty much in line with our description on performing a face rotation in section 4.6.1.1.

However, to perform a U or D move, we must perform an X or Z rotation first. To perform an X rotation:

1. Tell Brick FBClamp to release ClawF and ClawB
2. Tell Brick RLRotate to rotate ClawR clockwise 90 degrees and ClawL anticlockwise 90 degrees
3. Tell Brick FBClamp to close ClawF and ClawB
4. Tell Brick RLCclamp to release ClawR and ClawL
5. Tell Brick RLROtate to rotate ClawR anticlockwise 90 degrees and ClawL clockwise 90 degrees
6. Tell Brick RLCclamp to close ClawR and ClawL

We can now perform a U or D move by moving ClawF or ClawB.

### 4.6.3 Software

Now that we know about how the robot is set up, let's dive into the software. As mentioned in section 3.1.3.3 we use Lejos firmware for our implementation. This brings a few advantages over the previous BrickPi implementation: Lejos offers a fuller collection of libraries than the standard BrickPi libraries. These include: built in PID controller and PID-like controller interfaces, more online support and more example code. We've found that the default Lejos PID parameters provided in firmware version 0.9.1beta-3 give accurate enough motor control for this use case.

#### 4.6.3.1 Robot Class

We decided to implement a Robot Class to be used by the Android application. The Robot Class is responsible for connecting and sending appropriate messages to each of the NXT slaves. We represent a Robot as being comprised of 4 independent arms. Even though we chose to segment by job, the idea is to abstract this implementation detail and the bluetooth protocols away from the main implementation. This acts as the Robot Controller Layer in the design shown in Figure 3.2. To move any face, we simply call the `public void move(Move move)` method. Within this method, we simply control each of the arms. For example if we want to perform an R move:

Figure 4.23: move method

```
1 public void move(Move move) {
2     switch(move) {
3         ...
4         case R:
5             arms[RIGHTLEFT].rotateFace90(true, Arm.RIGHT);
6             break;
7         ...
8     }
9 }
```

The first parameter `true` tells us to rotate clockwise, the second parameter tells us which arm to rotate. In this case, since we are performing an R move, we pass `Arm.RIGHT`.

#### 4.6.3.2 Communication protocol

Within the Arm we have our Bluetooth communication protocol. The Arm class is responsible for sending bluetooth messages to each of the NXT Bricks. To create an Arm, we need 2 things: the Bluetooth address of the Brick responsible for clamping and the Bluetooth address of the Brick responsible for rotation. We use a series of protocols to communicate between the Smartphone and the NXT Bricks.

**Connecting Protocol** When the Arm is first constructed, we connect to the Bricks responsible for clamping and rotation. We then send a message to each Brick telling it what its job is: clamping or rotating.

**Move Protocol** Our move protocol allows us to perform moves in sequence synchronously or allows us to move multiple faces asynchronously. Moves are defined as follows:

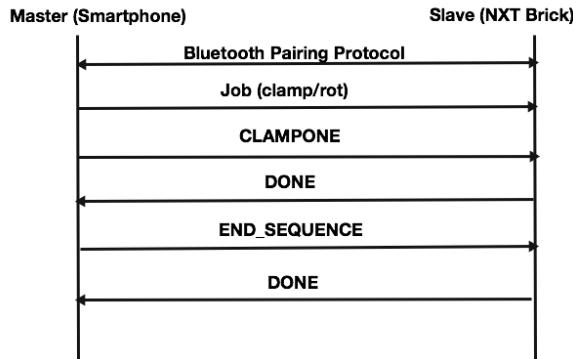
Figure 4.24: Moves

```

1 public Arm{
2     public static final int CLAMPONE = 0;
3     public static final int CLAMPTWO = 1;
4     public static final int UNCLAMPONE = 2;
5     public static final int UNCLAMPTWO = 3;
6     public static final int CLAMPBOTH = 4;
7     public static final int UNCLAMPBOTH = 5;
8
9     public static final int CLOCKONE = 6;
10    public static final int CLOCKTWO = 7;
11    public static final int ANTIONE = 8;
12    public static final int ANTITWO = 9;
13    public static final int CLOCKBOTH = 10;
14    public static final int ANTIBOTH = 11;
15    public static final int CLOCK180ONE = 12;
16    public static final int CLOCK180TWO = 13;
17    public static final int ANTI180ONE = 14;
18    public static final int ANTI180TWO = 15;
19
20    public static final int DONE = -1;
21    public static final int END_SEQUENCE = -777;
22    ...
23 }
```

The suffix ONE tells the Brick to perform the move specified in the prefix using the first motor port. Analogously, TWO tells the Brick to use the second motor port. Finally, BOTH means perform the prefix action using both motor ports. We send one of these messages to the corresponding Brick. If the Brick's job is to clamp and we send it a rotation command, it will simply ignore it. Otherwise, the Brick will perform the action specified. Once the Brick finishes the movement, it will send a DONE message back. We can either choose to wait for this message or continue. This is how we achieve the synchronous or asynchronous behaviour. Once we finish transmitting all the moves we need, we send an END\_SEQUENCE message to all of the Bricks to tell them to reset their positions ready for the next solve.

Figure 4.25: An example of the whole protocol



#### 4.6.3.3 Motor rotation parameters

Every motor is built slightly differently so one motor might not necessarily have the same behaviour as another even if they are given the same parameters. Each motor needs to be calibrated individually. In this section we detail our calibration process and approximate parameters for each command.

**4.6.3.3.1 First Approximation** We calculate the first approximation using the following formula:

$$\text{Degrees To Move} = \frac{N_{\text{secondary}}}{N_{\text{main}}} * \text{desired\_angle} \quad (4.6)$$

where  $N_{\text{main}}$  is the number of ‘notches’ in the main gear attached to the motor and  $N_{\text{secondary}}$  is the number of ‘notches’ in the second gear next to the main gear. Figure 4.26 shows the gears used for controlling arm rotation. The number of notches for the main gear is 44 and the secondary gear is 60. We can calculate the how many degrees we require our motor to move for a 90 degree rotation as follows:  $60/44 * 90 \approx 123$ .

Figure 4.26: The rotational gear ratio



**4.6.3.3.2 Adjustment** Although we have calculated how many degrees we wish to turn, this is only an approximation. In reality, we need to adjust this. To adjust this parameter, we simply perform the moves on a Rubik’s cube and check for under/over rotation and adjust the parameters accordingly. We’ve found that our parameters vary at most 3-4 degrees between motors.

Figure 4.27: Degree rotation table

Command	Degree	Variation
Clamp	-95	$\pm 2$
Unclamp	95	$\pm 0$
Clockwise 90 Rotation	-123	$\pm 2$
AntiClockwise 90 Rotation	123	$\pm 2$
180 Degree Rotation	247	$\pm 1$

# Chapter 5

## Evaluation

### 5.1 Vision

#### 5.1.1 Vision accuracy

We've seen how our vision system is implemented, but how accurate is it? In this case, we measure accuracy by the number of stickers the vision system correctly recognises. There are two parts to the vision system that need to be measured here: the cube recognition process and the colour recognition process. It is important that both of these obtain high scores in accuracy. The cube recognition process is the less important of the two. After all, if the video camera API is feeding in 6 or 7 frames every second, we will get lots of chances to try to recognise the cube from each frame. More serious errors occur when we wrongly identify a colour since the system will not know it has wrongly identified a colour. This could then lead to impossible cube states or wrong solutions.

##### 5.1.1.1 Testing Lighting Environment

In this test, we take our vision system and attempt to read cube states in various lighting scenarios. For each lighting scenario, we take 20 photographs of different cube states with varying backgrounds and at varying distances and track 3 things:

- The number of correctly recognised colours: passes.
- The number of wrongly recognised colours: failures.
- the number of failures to find the cube in the photograph: errors.

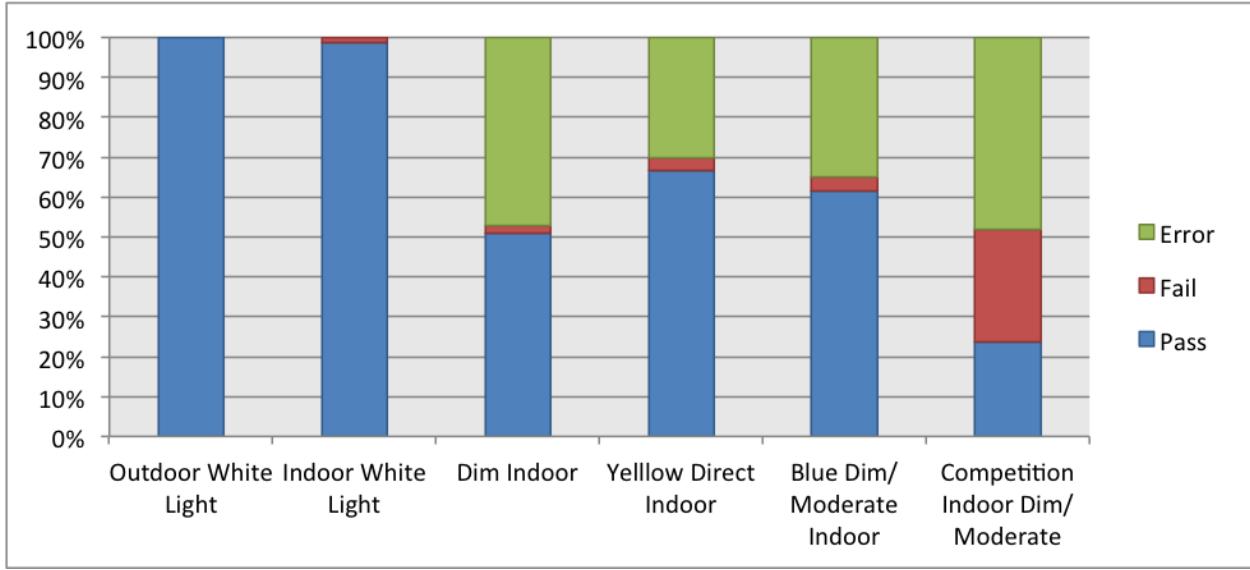
Not being able to find the cube in the photograph is penalised the same way as getting every sticker of a face wrong so number of errors moves in increments of 9. The other two items of data move in increments of 1 for each sticker they recognise correctly or incorrectly. As well as scenarios, we have compared our vision system with that of Yakir Dahan and Iosef Felberbaum [6] which is well documented and has a version of their application on the Android Play Store <sup>1</sup>.

The results are as follows:

---

<sup>1</sup>At the time of writing, the latest version is 1.0

Figure 5.1: Test Results For Vision



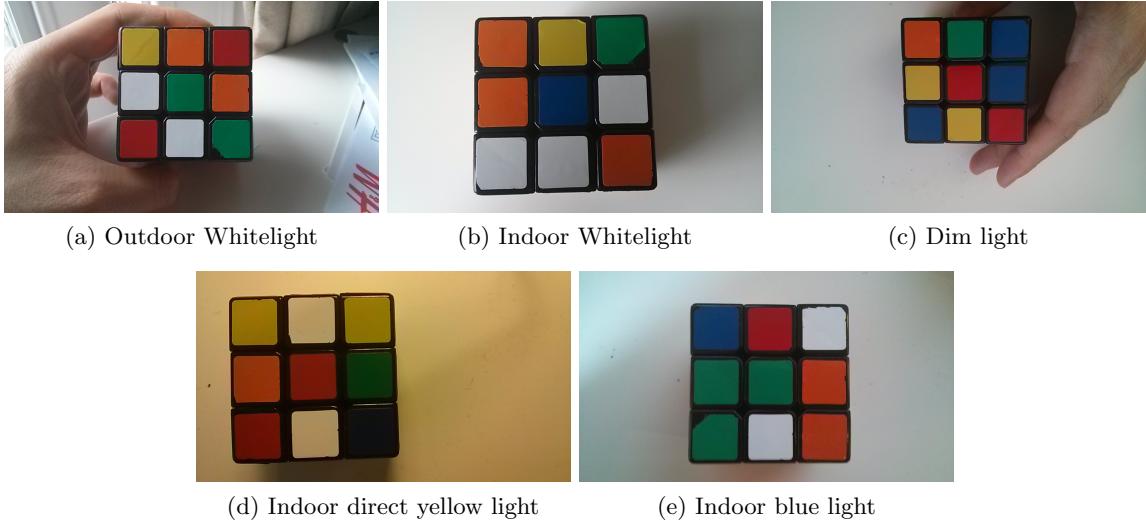
We can see that when we have outdoor white light, the cube is easily visible and the contours of the cube are easily recognisable. In fact, we have 0 errors and 0 failures when we have sufficient light to illuminate the cube i.e. we recognise it correctly in all photos. Similarly, indoors, we have 0 errors and just a small percentage of colour recognition failures (less than 1%).

In dim scenarios, we start to see a major drop off in reliability and robustness. We found that most of the pictures taken by our test device (Nexus 4) in low light scenarios were grainy and out of focus. This could have contributed to the dramatic decline in our pass count. Colours also become harder to distinguish when there is less light. However, our pass rate of over 50% is significantly higher than the 25% measured with the competition. This shows that our cube recognition process is more robust than the competition under low light.

If we now move towards different coloured illuminations, we chose to use the most common indoor lighting scenarios: yellow light and blue light. We can see that under direct yellow light (In this case we used a lamp with a fluorescent bulb) our vision system still performs reliably with almost a 70% pass rate. Our error count does increase when compared to outdoor or indoor white light, however. This may be caused by heavy shadows being cast on the cube under direct light which makes it harder to find contours. We also see that our error count is still quite low which means that our white balancing algorithm is working nicely here.

Under blue light, we see similar results to yellow light. Our lighting conditions are slightly different. Our light was brighter than our dim test but not as bright as our indoor white light test. Our error percentage rises slightly due to low light. With the success rate still above 60% and a low colour recognition failure percentage, the system is still perfectly usable under dim/moderate blue light. Below we have included some sample images of each lighting scenario:

Figure 5.2: Sample images of each light scenario



### 5.1.2 Vision limitations

Although we have covered many areas of the vision system to try and make it more robust and reliable, there will always be some areas which we can improve upon.

#### 5.1.2.1 Glare and Darkness

Although we have accounted for many types of lighting, our implementation falls short when there is excessive glare caused by direct light (such as lamps). The glare makes regions of the cube appear white when in fact they are not. This can then affect the edge detection as well as the colour recognition process. Since this was an unlikely scenario (most rooms have a ceiling light or windows that diffuse the light pretty well), we decided to ignore this. The current workaround is to attempt to diffuse the light. This can be done by directing the light source (e.g. if you are using a lamp) towards a wall and using the diffused light to illuminate the cube.

Conversely, a dimly lit room can cause an equal amount of inaccuracy. In a dimly lit room, the stickers are difficult to differentiate from the cube. The cube is also difficult to differentiate from the background. In a dark room, we lose a lot of information on the edges and the sticker colours. With our test device (Nexus 4), the camera lacked Optical Image Stabilisation and used a lens with a small aperture. Both limit the amount of light entering the camera, giving the impression of a really dark image. There is not much we can do in this situation other than attempt to introduce some light into the environment.

#### 5.1.2.2 User Intervention

Although the main focus of this project was not user friendliness, requiring human intervention to physically have to rotate the cube under the camera is not ideal. It is slow and prone to human error since the vision system assumes the cube will always be moved in a specific direction and in a specific orientation. Divergence from this will cause problems when trying to build the cube. We want to improve on this by having the robot move the cube for us under the camera which would then eliminate all of these human related issues.

#### 5.1.2.3 Gray World Assumption

The problem with the Gray World Assumption is that it assumes the average colour of the image is grey. This is the case when we have an image with various colours. In nearly all cases, a Rubik's cube will have

an even spread of colours on each face. But what about cases where it doesn't? In the rare cases such as this case:

Figure 5.3: A rare case



In this case, the average colour of the image is red. Gray World Assumption would mistakenly think that the image is illuminated from a red light source! Our current work around for this is having a button in the application that allows the user to enable or disable white-balancing. The user can then decide if the image needs white balancing or not themselves. This obviously isn't the best solution but for now the user can continue to use the application until we find a more robust colour balancing algorithm.

#### 5.1.2.4 OpenCV

Although OpenCV gives a lot of functionality it also leaves a lot to be desired. In order for the application to work, we must also download the OpenCV Manager application to supplement the use of the OpenCV libraries. The problem with this is that the OpenCV Manager lacks compatibility with a long list of Android devices making it harder for us to test across multiple devices. Additionally, there is a known bug with the Nexus 4 camera and OpenCV version 2.4.6 where the default camera only gives a maximum frame rate of around 10 frames per second on the lowest resolution 320x240 when other slower devices are able to get 30 frames per second. It also causes the device to randomly reboot.

#### 5.1.2.5 Colour Schemes

Our vision system works with the Western Colour Scheme (BOY)[9]. This means that green and blue are opposite, yellow and white are opposite and orange and red are opposite. This is because we assume that the U face is blue, D is green, F face is white, B face is yellow, R face is red and L face is orange.

#### 5.1.2.6 Vision Summary

Our original goal was to make a robust and reliable system that would be able to read the state of the cube quickly. From the tests we've seen above, in common lighting scenarios, that our system is both robust and reliable with the worst case being 50% pass rate in a scenario with dim lighting. Although this may seem low, in reality, our application's video frame rate is around 6-7 frames per second which means that we are very likely to succeed in reading a face state within that second. Our chance of failure within this second is  $\approx (0.5)^7$  which is less than 1% chance! We are also able to read the cube state at varying distances from the camera and with objects in the background. We've found that it takes less than a second to read the cube state completely the vast majority of the time and the process is mostly bottlenecked by us needing to physically rotate the cube for each face.

Given that this is just one part of the bigger project, we are overall very happy with the way the vision system performs although there is obviously room for improvement.

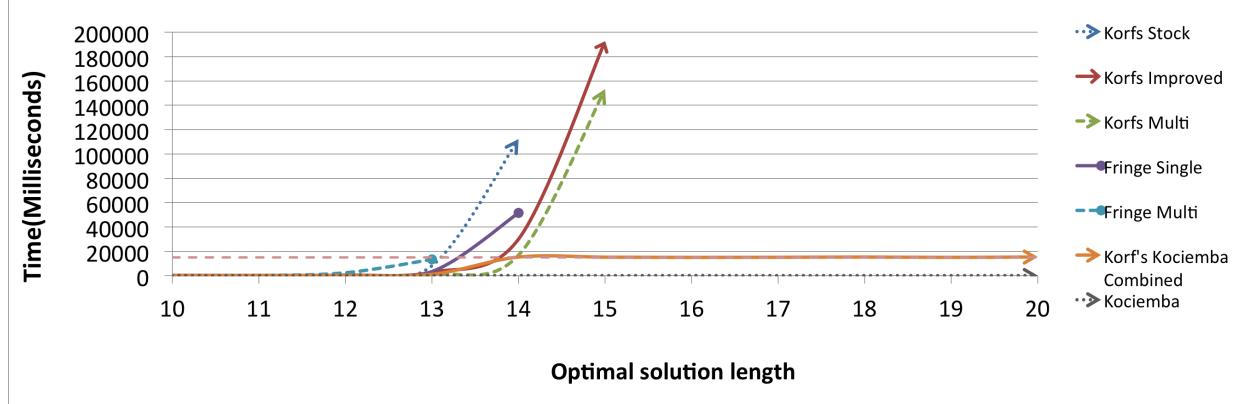
## 5.2 Algorithm

In this section we want to benchmark our algorithm against existing ones. For these benchmarks we use a dual-core Haswell Intel Core i5-4258U CPU 2.4GHz. We have configured our Java VM using 2 flags: `-Xms2048m`, `-Xmx6144m` to give the VM enough Heap space so that our algorithm speed tests will not be bottlenecked by swapping.<sup>2</sup>

### 5.2.1 Algorithm speed

In this test we aim to measure how fast our algorithm can compute a solution (regardless of how close the solution is to optimal). We will prepare solves with an optimal solution length varying from 10 to 20 and take the average time using various algorithms. We will also do the same for the standard Kociemba's algorithm and standard Korf's algorithm. If the solve takes longer than 5 minutes, we deem the test as failed and do not plot any further results.

Figure 5.4: Speed of algorithms in Milliseconds



The lines are labelled as follows:

- **Korfs Stock** represents the original Korf's algorithm with no improvements. (Blue dotted line)
- **Korfs Improved** represents Korf's algorithm with all of our improvements excluding parallelism and fringe searching. (Red solid line)
- **Korfs Multi** represents Korf's algorithm with all of our improvements including parallelism but excluding fringe searching. (Green dashed line)
- **Fringe Single** represents Korf's algorithm with all of our improvements excluding parallelism but including fringe searching. (Purple solid line)
- **Fringe Multi** represents Korf's algorithm with all of our improvements including parallelism and fringe searching. (Blue dashed line)
- **Korf's Kociemba Combined** represents our Korf's and Kociemba combined algorithm. (Orange solid line)
- **Kociemba** represents the original Kociemba's algorithm given by Kociemba's libraries. (Grey dotted line)

<sup>2</sup>Thanks to Rokicki, Romas and Kociemba, Herbert and Davidson, Morley and Dethridge, John @ [www.cube20.org](http://www.cube20.org) for their list of known 20 move scrambles

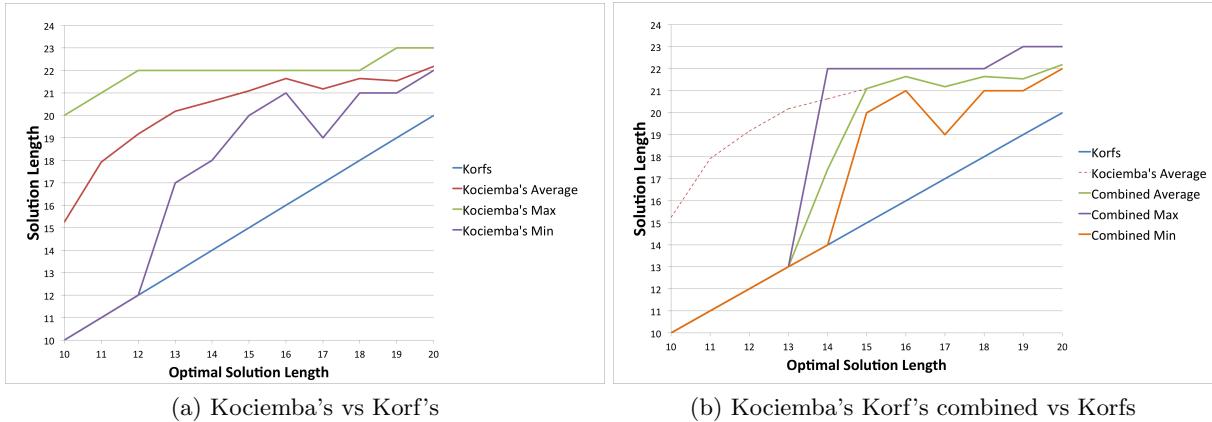
We can see that all algorithms cope fairly well until a depth of around 13. Perhaps the most surprising results are Single/Multithreaded Fringe Searches We found that anything above a depth of 13 would make our fringe lists use far too much memory (greater than 6GB) and the whole system would then be bottlenecked by Java's Garbage collector. Additionally, we found that in our multithreaded implementation of fringe search, the overhead of synchronising locks to our fringe list exceeded any benefit we received from having multiple parallel searches. Perhaps a better implementation would have been to use  $n$  mutually exclusive fringe lists that each thread can manage on their own. This way, we would not need to synchronise any access. Each thread would then be responsible for performing a fringe search for its selected portion of the fringe.

Amazingly, we see that there is a big improvement of the original stock Korf's algorithm vs Korf's Improved algorithm (with improvements mentioned in section 4.1.3. Our improvements allowed us to search an entire extra depth in almost the same time. Unsurprisingly, the biggest speedup came from our Multithreaded IDA\* algorithm. We managed to obtain this speedup using only 2 cores. We suspect we would see even bigger speed ups with 4 or even 8 cores. The dashed red line shows the 15 second mark that solvers are allowed to have for inspection before attempting a solve. We can see that our multithreaded IDA\* algorithm just about makes the mark for a depth of 14. Anything above depth 14 takes significantly longer than 15 seconds. At around this mark, we can see that our Korf's and Kociemba combined algorithm starts to make a switch from Korf's to Kociemba's solutions since Korf's is using all of its allotted 15 seconds. Predictably, Kociemba's algorithm remains consistently under 15 seconds. In terms of speed, Kociemba's algorithm clearly wins out for large depths. Although not quite visible in the graph, we did find that Korf's Improved and Korf's Multithreaded were able to beat out Kociemba's algorithm for depths below 12.

### 5.2.2 Algorithm solution length

In most cases, a shorter solution is preferred. A shorter solution will give the robot less 'work' to do in most cases, which reduces the time of the solve. In this section we compare Korf's, Kociemba's and Korf's Kociemba's combined for their length of solution. (Other algorithms are just variations of Korf's and would give the same length solution). We took scrambles with optimal solutions of varying length 10-20 and found the length of solution given by each algorithm. We took the minimum, mean and maximum for each algorithm.

Figure 5.5: Solution Lengths



On the left we can see that Kociemba's algorithm becomes more consistent the closer it is to solving a 20 move scramble. Solves that require 10 - 14 moves saw massive variation with a maximum difference of 10 moves away from the optimal solution. On average, Kociemba's comes pretty close to optimal once we get above 17 move solves - requiring 5 moves more than optimal at most. Obviously this is just a sample of solves. In reality, it has been said that Kociemba's algorithm can give a maximum of 30 moves (although this is rare).

On the right we can see our Kociemba's Korf's combined algorithm. As expected, up until around a 14 move scramble, we remain optimal; a big improvement over Kociemba's algorithm. The 'average' line begins to shift towards Kociemba's algorithm's average (red dotted line) at around 14. At this threshold, we had some solves that were able to be solved under the given 15 seconds, others that were not. This brought down the average slightly over just using Kociemba.

### 5.2.3 Algorithm Summary

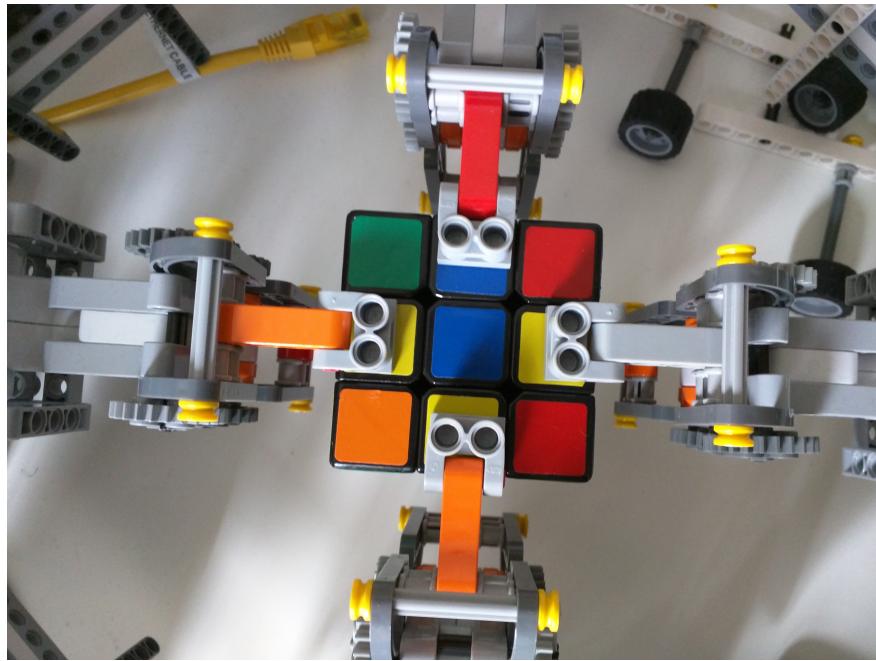
Our goal was to have an algorithm that would be as close to optimal as we could get. We were able to bring Korf's algorithm's search times down to stay competitive with Kociemba's algorithm up to a depth of around 14 on a dual core machine by using our improvements from section 4.1.3. This is an improvement over stock Korf's algorithm which could only stay competitive up to a depth of around 13 on our machine. Our combined algorithm was able to improve on Kociemba's algorithm at lower depths where Kociemba struggled to not only give an optimal solution, but also struggled to give a solution of consistent length. However, our limitation is that we are still not able to get an optimal solution for every random scramble. Additionally, we learned that although a fringe search seems like a good idea in theory, for this particular application, the overhead of managing large lists proved to be too much to make the fringe search usable. Overall, we did not get the speed-ups we expected from all of our optimisations, but we did learn a lot of valuable lessons, especially about fringe searching.

## 5.3 Robot

### 5.3.1 Robot Accuracy

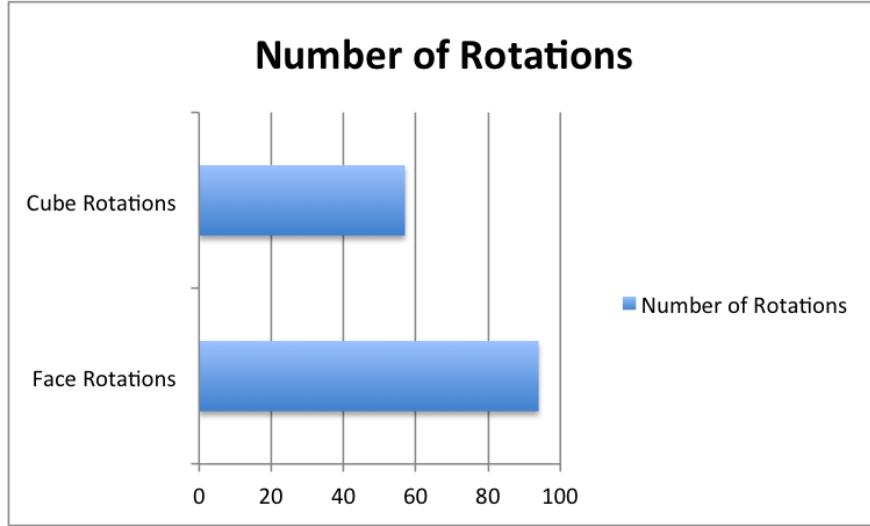
In this test, we perform numerous face rotations and physically measure how far the face is away from a perfect rotation. Any inaccurate face turning that will cause subsequent moves to fail will be deemed a failure. Once we reach failure, we will stop testing any further. The more moves the robot can make, the better it will score. Figure 5.6 shows what a perfect alignment of claws and faces looks like.

Figure 5.6: A perfect alignment



We also perform cube rotations and measure how far the clamps are off centre for each cube rotation. Anything that will impede subsequent moves will be deemed a failure.

Figure 5.7: Number of face turns and cube rotations until failure



We can see that the PID controller really helps with achieving the desired accuracy. We were able to perform 90+ face rotations before the drifts caused a failure. Since each solution length can only be 30 at most and on average around 20, our chance of failing during a solve will be quite low! In fact, over 5 trials, there were no instances where the face rotation number did not exceed 70 before failure.

Cube rotations on the other hand are seemingly less reliable. We found that on average we were able to perform around 50 cube rotations before failure. Although this is much lower than face rotations, the number of cube rotations in a solve on average is significantly less than face rotations. If we look at an average case where we assume each move is equally likely to happen in a 20 move sequence then 1/3 of the time we will need a cube rotation. This means on average we require around 6-7 cube rotations for a solve. Over 5 trials, there were no instances where the number of cube rotations did not exceed 30 before a failure. On average, the cube rotations should be reliable enough for every solve.

### 5.3.2 Robot Speed and TPS

In this test, we perform 7 random solves and time how long it takes to solve so that we can take the average face TPS (turns per second). A 180 degree move still counts as just 1 turn. We start the time as soon as the first move is made. Below are our results:

Figure 5.8: Solve times table

Length of Solution	Time to solve (Seconds)	TPS
20	70	0.29
22	85	0.26
21	74	0.28
20	65	0.31
20	69	0.29
20	75	0.27
22	81	0.28

We can see that on average, our solve times lie around 74 seconds. Our average turns per second is just 0.28.

### 5.3.3 Robot Limitations

#### 5.3.3.1 Claw Slippage

Naturally, any robot with no external facing sensors will experience some sort of drift. Although our cube rotations are quite reliable, there is the odd occasion where we perform a cube rotation and the cube slips between the claws. This is because during the rotation, only 2 claws will be holding the cube (as described in section 4.6.2). Even a small amount of slippage can cause an adverse effect on the rest of the solve. The robot will not know this has happened and continue to try to solve the cube. The cube will no longer be centred and we will not be able to finish the solve.

#### 5.3.3.2 Gear Slippage

Lego gears do not align perfectly with each other. It is natural that there can be some gear slippage where the ‘notches’ on the gears become disconnected because they are moving too fast. This is particularly prominent in the claw grabbing mechanism where the gearing system had to be compact in order to rotate quicker. Gear slippage throws off the whole solve. The gears are aligned in the claw so that they grab the cube perfectly. Any deviation will cause the claw to loosen its grip which can lead to claw slippage as described above.

#### 5.3.3.3 Speed vs Accuracy

We can see that our solve times are not particularly fast even given a relatively short solution. The culprit is our TPS; it is too low. Here a decision had to be made between accuracy and speed. The more we increased the speed of the system, the less accuracy we would have. This is mostly prominent in cube rotations where the inertia of turning the cube too fast would cause the cube to ‘jolt’ and cause claw slippage. This is why we made the decision to tone down the speed in favour of reliability.

### 5.3.4 Robot Summary

Our goal was to create a robot that would be able to reliably solve the Rubik’s cube. Thanks to Lejos’s PID controller, we were able to achieve acceptable accuracy to turn the cube enough times to solve it. Our robot controller interface also allows us to control the robot wirelessly with ease. However, the design is still not perfect and is susceptible to drift and claw slippage. Our TPS could also be improved.

### 5.3.5 System

#### 5.3.5.1 System Limitations

The main limitation with our system is that it isn’t a closed system. We require human intervention during the vision phase and we also require a PC to search for solutions. In a closed system, the vision would be autonomous because the robot would be able to move the cube to view each side. We also need the user to place the cube in the correct position for the claws to grab the cube. If this is off centre, the user must manually adjust the position of the cube within the claws. Additionally, the system is not bullet-proof. Each component has its own limitations which in the end brings down the robustness and reliability of the system as a whole.

# Chapter 6

# Conclusions

We have built a system capable of solving the Rubik's cube within 70 seconds. Our system uses a reliable and robust vision system, a fast and close to optimal solution finding algorithm and an accurate 4 armed Lego robot. Our vision system is able to cope well under various lighting environments and can perform well even with background noise. Our algorithm can find optimal solutions in a reasonable time for solution depths of up to 14 and also give solutions that are close to optimal for depths above 14. Our robot can solve a Rubik's cube quickly and reliably enough to rival intermediate level speedsolvers.

Although we weren't hoping to break any world records, we saw this instead as an opportunity to explore and compare many different aspects of each component of the system. We have compared and documented various search algorithms, vision algorithms and robot builds for solving the Rubik's cube which has never before been done in this field. There have been Rubik's cube solvers built in the past, but most of them use a 'canned' stock Kociemba's algorithm and are rarely documented in such detail.

## 6.1 Future work

In this section, we discuss the areas that we feel can be improved in each of the components in the system. We offer insight into what we would have liked to implement had there been more time for this project.

### 6.1.1 Improving the Algorithm

#### 6.1.1.1 Interleaving Fringe Search in Korf's

Although our Fringe Search implementation didn't work as expected, we suspect that this may have been due to the overheads associated with managing the fringe lists. We can reduce the size of the fringe lists if we work in parallel with IDA\*. That is, if we start an IDA\* search but a portion of the tree is then reserved for fringe searching to take place on another thread. Only a small section of the tree is given to the fringe searching thread so the fringe list should remain relatively small. Once we finish the fringe search on one portion of the tree, we can ask the thread performing the IDA\* search for more work. The IDA\* thread can then give the fringe search thread another portion of the tree that it may be halfway through searching to finish off. This method of fringe search interleaving would hopefully speed up searching because the fringe search would finish off these smaller portion searches a lot quicker than IDA\* would.

#### 6.1.1.2 Better Dynamic Heuristics

Korf's algorithm currently uses a very natural way of segmenting the cube into substates: corners, 6 of 12 edges, and other 6 of 12 edges. We would like to explore other heuristics where we could mix and match corners and edges. For example, 4 corners and 4 edges, or only look at corner orientation and edge orientations. Ideally, we'd like to find substates with pattern databases that fit perfectly into the memory

of our system. This is the dynamic aspect: we choose which substate heuristics to use depending on the system's memory capacity. This would make use of the memory that IDA\* saves.

#### **6.1.1.3 Symmetry Reduction**

In addition to better heuristics, we would like to experiment with using Kociemba's idea of Symmetry Reduction[14] in Korf's algorithm. Symmetry Reduction uses equivalence classes to reduce the sizes of our pattern databases. In a nutshell, 2 cubes are equivalent if we can rotate the cube and recolour the stickers of one and obtain the state of the other. For example, imagine performing an R move from the solved state. Imagine another cube that has a B move performed from its solved state. If we perform a Y rotation, and recolour the second cube, it is equivalent to the first so we don't have to store entries in the pattern database for both of these states. Using Symmetry Reduction, we can reduce the size of our pattern databases which also gives us more room for improving our heuristics.

#### **6.1.1.4 Probabilistic Search**

Currently, we explore each branch with equal probability but this is never the case when a cube is being scrambled. In fact, when a human scrambles the cube, they tend to prefer turning specific sides more than others. We would like to implement some form of probabilistic searching algorithm based on previous solutions. This will allow us to explore most likely branches first and potentially speed up the search for a solution. This would only work if a human scrambles the cube. If we were to use a computer generated scramble, then the scrambling would be random so we wouldn't benefit from this technique.

#### **6.1.1.5 Multithreaded IDA\*/Fringe in Kociemba's**

Kociemba's libraries provided us with a working implementation of his algorithm. This saved us a lot of time so that we could focus on other areas of the project. Given more time, we would have liked to have tried our variations of Korf's algorithm in the search portion of Kociemba's algorithm. In theory, since the search depths are lower, fringe search should be more effective here.

### **6.1.2 Improving vision**

#### **6.1.2.1 More Robust Colour Recognition**

Our current implementation of colour recognition is quite reliable. However, it starts to become less usable in dim lighting scenarios. We would like to implement automatic brightness and contrast features into the vision application so that we can better cope with these situations.

#### **6.1.2.2 Autonomous Scanning**

The main bottleneck in our system is that we need the user to rotate the cube so that the camera can view each side of the cube. We want to eliminate this bottleneck by having the robot rotate the cube instead. The camera would then lie in a fixed position over the cube and view each side quickly. This would also reduce the chance of human error and we would also be able to make assumptions about the background since the background will always be fixed.

### **6.1.3 Improving the Robot**

#### **6.1.3.1 Gear Slippage Reduction**

So far, the biggest cause of errors during a solve is gear slippage. We would like to reduce this by reinforcing the gear train mechanism on the claw. This would reduce the gear slippage and give us a more reliable solve.

### **6.1.3.2 Rotation Speed**

Currently, our rotation speed is quite slow. We made the decision to sacrifice some rotation speed in favour of stability and accuracy of turning. The biggest problem caused by high turn speed is vibration and inertia. Vibration can throw off the alignment of the claws and inertia can jolt the cube out of alignment. We would like to improve our turn speed and in order to do so, we would need to build a bigger and more stable structure that isn't as susceptible to vibration. To solve our inertia problem, we can look into PID controller adjustments so that we can dampen the deceleration of the motor more. We should also look into our claw design and find ways to get more grip on the cube.

### **6.1.3.3 Concurrent Rotation**

Again, in favour of stability, we decided that it is best to only have one face turn at a time and have the 3 remaining claws grip the cube. We found that 2 claws gripping the cube wouldn't provide enough stability to keep the cube stationary. We would like to improve the gripping mechanism so that we can turn opposite faces concurrently and also begin to turn adjacent faces as soon as a face turn is complete (before the reclamping).

# Bibliography

- [1] How to find the index of a k-permutation from n elements? <http://stackoverflow.com/questions/24215353/how-to-find-the-index-of-a-k-permutation-from-n-elements>. Accessed: June 1, 2015.
- [2] Mindcuber. <http://mindcuber.com>. Accessed: June 1, 2015.
- [3] JP Brown. Cubesolver. <http://jpbrown.i8.com/cubesolver.html>. Accessed: June 1, 2015.
- [4] Janet Chen. Group theory and the rubik's cube. page 11.
- [5] Joe M. Converse. Basic notation. <http://w.astro.berkeley.edu/~converse/rubiks.php?id1=basics&id2=notation>. Accessed: June 1, 2015.
- [6] Yakir Dahan and Iosef Felberbaum. Rubik's cube solver. *Efi Arazi School of Computer Science*, pages 3–4, 2014.
- [7] Andrew Davison and Stefan Leutenegger. Lecture 2: Robot motion. PID Controllers, 2015, Slide 20.
- [8] Pasquale D'Silva. A little about color: Hsv vs. rgb. [http://www.kirupa.com/design/little\\_about\\_color\\_hsv\\_rgb.htm](http://www.kirupa.com/design/little_about_color_hsv_rgb.htm). Accessed: June 1, 2015.
- [9] Dnes Ferenc. The rubik's cube colour schemes. <http://ruwix.com/the-rubiks-cube-japanese-western-color-schemes/>. Accessed: June 1, 2015.
- [10] Matthew Hatem, Burns Ethan, and Rumi Wheeler. Faster problem solving in java with heuristic search. *developerWorks*, pages 12–16, 2013.
- [11] Ryan Heise. Rubik's cube theory: Laws of the cube. [http://www.ryanheise.com/cube/cube\\_laws.html](http://www.ryanheise.com/cube/cube_laws.html). Accessed: June 1, 2015.
- [12] Ryan Heise. Rubik's cube theory: Parity. <http://www.ryanheise.com/cube/parity.html>. Accessed: June 1, 2015.
- [13] Herbert Kociemba. The facelet level. <http://kociemba.org/cube.htm>. Accessed: June 1, 2015.
- [14] Herbert Kociemba. The two-phase-algorithm. <http://kociemba.org/cube.htm>. Accessed: June 1, 2015.
- [15] Richard E. Korf. Finding optimal solutions to rubiks cube using pattern database. *Computer Science Department, University of California*, pages 2–4, 1997.
- [16] OpenCV. Laplace operator. [http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/laplace\\_operator/laplace\\_operator.html](http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html). Accessed: June 1, 2015.
- [17] pi19404. Automatic white balance algorithm. <http://www.scribd.com/doc/117031630/automatic-white-balance-algorithm-1#scribd>. Accessed: June 1, 2015.
- [18] Romas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. God's number is 20. <http://www.cube20.org>. Accessed: June 1, 2015.

- [19] Jaap Scherphuis. Thistlethwaite's 52-move algorithm. <http://www.jaapsch.net/puzzles/thistle.htm>. Accessed: June 1, 2015.
- [20] Martin Schner. Analyzing rubik's cube with gap. <http://www.gap-system.org/Doc/Examples/rubik.html>, 1993. Accessed: June 1, 2015.
- [21] David Singmaster. *Notes on Rubik's Magic Cube*. Enslow Pub Inc, 1981.
- [22] Mikhail Vorontsov. Java performance tuning guide: Memory consumption of popular java data types - part 2. <http://java-performance.info/memory-consumption-of-java-data-types-2/>. Accessed: June 1, 2015.
- [23] Wikipedia. Berkeley algorithm — wikipedia, the free encyclopedia, 2015. [Accessed: June 7, 2015].
- [24] Wikipedia. Cubestormer 3 — wikipedia, the free encyclopedia, 2015. [Accessed: June 7, 2015].
- [25] Wikipedia. Factorial numbering system — wikipedia, the free encyclopedia, 2015. [Accessed: June 7, 2015].
- [26] Robert C. Holte Yngvi Bjornsson, Markus Enzenberger and Jonathan Schaeffe. Fringe search: Beating a\* at pathfinding on game maps. pages 2–3.

# Appendix A

# System User Guide

## A.1 Prerequisites

### A.1.1 Hardware requirements

Our application requires an Android Smartphone with a minimum of:

- Android 4.0.3 (API 15)
- Bluetooth
- Camera
- WiFi

The smartphone must also support OpenCV manager. So far, we have only tested our application using OpenCV Manager version 2.18.

### A.1.2 Setup

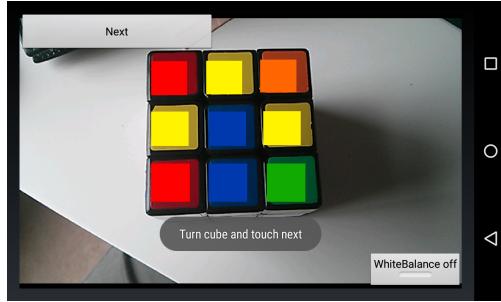
We need to set up a few things before we can use the system. Firstly, we must pair all NXT Bricks with our Android Smartphone via Bluetooth and ensure WiFi and Bluetooth are turned on. Next, we need to ensure our PC is ready to receive cube states so that it can find solutions. We can start a local server instance from our PC by using the main function inside of our RubiksSolver project in package com.rubiks.lehoang.rubikssolver. We then turn on all NXT Bricks and run the program Arm.nxj on each Brick.

## A.2 Using the system

### A.2.1 Reading the state

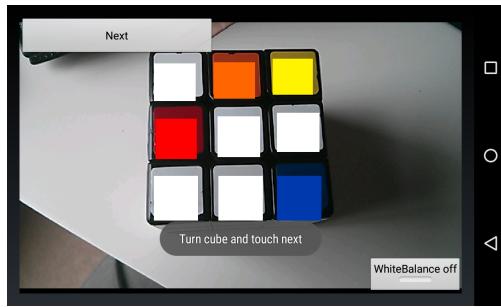
Firstly, we need to read the cube's state. We touch the 'Read State' button on the Smartphone to bring up the camera. The faces need to be taken in a specific order and orientation as follows. We start with a cube using the Official Western Colour Scheme[9]. The blue centre piece must be on the U face, white must be on the F face and red must be on the R face. We take a picture of the U face first. Hold the camera horizontally and turn on or off white-balance using the white-balance toggle as appropriate. When the program recognises the cube, it will display squares over each sticker of what it thinks the colours are, like so:

Figure A.1: U Face



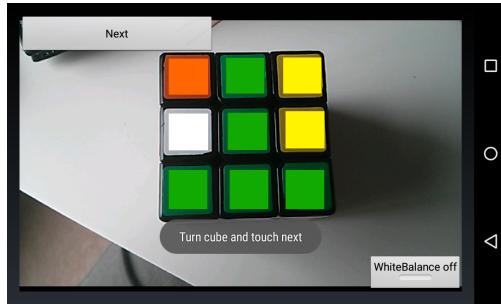
If the colours are correct, rotate the cube to the F face using an X rotation and click next. Otherwise, click next and try the same face again until the colours are correct. You can click next and try the same face as many times as you need.

Figure A.2: F Face



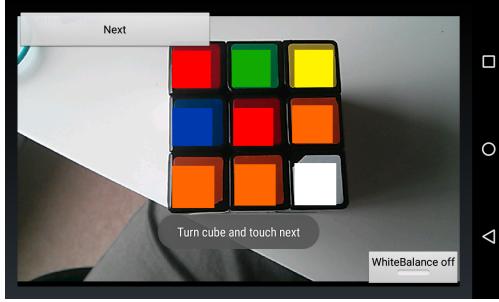
Once the app has taken the correct colours from the F face, perform another X rotation to the D face. Touch next and repeat the same steps again until the colours are correct.

Figure A.3: D Face



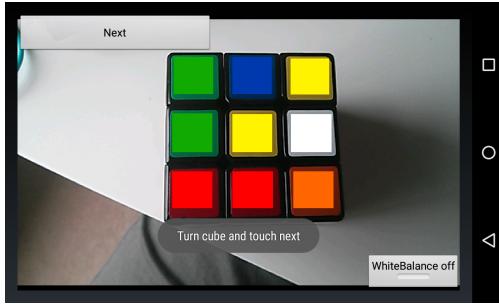
We then perform an X3 rotation followed by a Z rotation. This should get us to the R face. Touch next and repeat.

Figure A.4: R Face



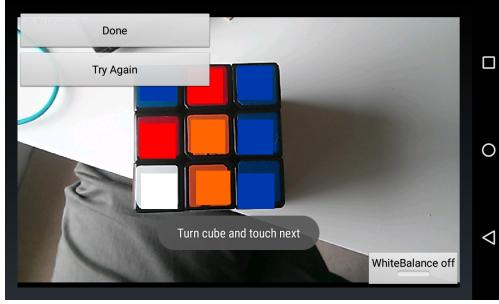
Perform another Z rotation to move to the B face. Touch next and repeat.

Figure A.5: B Face



Perform another Z rotation to move to the L face. Touch next and repeat again. Once we have the final face, we can either touch ‘Done’ if the colours are correct or ‘Try again’ if the L face was not recognised correctly.

Figure A.6: L Face

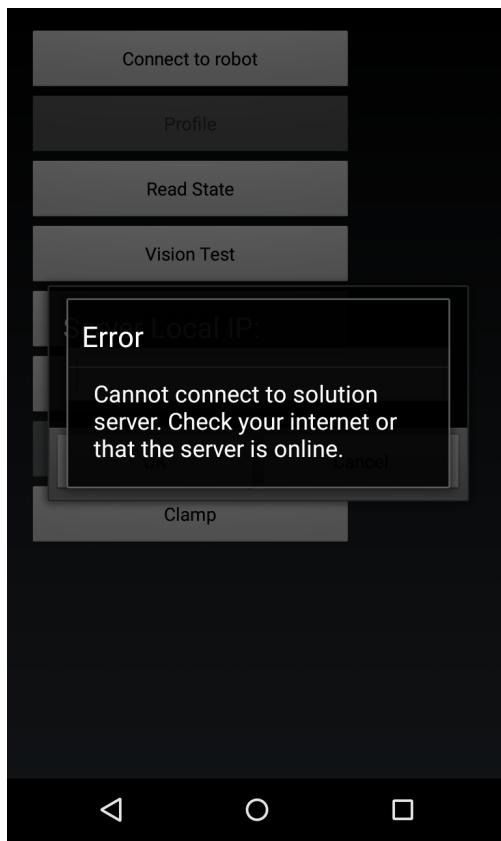


If we've performed this correctly, we should get a dialog box that says ‘We have a cube state!’. Otherwise, we will get an error message.

### A.2.2 Find a solution

Now that we have the cube state, we can find a solution. Ensure the Local Server instance of the solution finder is running and press ‘Find Solution’. On your first attempt, this should fail:

Figure A.7: First Connection Failure



Dismiss this message and enter the Local IP address of where your Local Server instance is running. If you've entered the correct IP, pressing ‘Find Solution’ again should be successful and give back a ‘Got solution!’ message.

### A.2.3 Solving the cube

Place the Rubik’s cube into the robot and ensure all NXT Bricks are turned on and running the `Arm.nxj` program. Touch ‘Connect to robot’ within the Smartphone application and wait for the device to connect to all of the Bricks. If you’ve already paired the Smartphone to all of the Bricks, this step should be quick. Otherwise, Android will prompt you to pair each device before you can start. Once connected, the clamps will automatically close and grab the Rubik’s cube. If it is not perfectly aligned with each face, make small adjustments to align the claws. Once aligned, click ‘Solve’ and watch the robot solve the cube! Once it has finished solving, the robot will release the cube.